

ACOUSTIC FINITE ELEMENT MODELING OF AN IMMERSED STRUCTURE

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Mija Helena Hubler

May 2009

© 2009 Mija Helena Hubler
ALL RIGHTS RESERVED

ABSTRACT

This work investigates an acoustic fluid structure interaction formulation. It is applied to an immersed steel structure in non-flowing water. The system is discretized using 3D solid continuum and acoustic fluid finite elements. Fluid structure interactions are formulated according to Zienkiewicz's work for submerged dams. We find that the sensitivity of various approximations and problem parameters are critical to fluid structure finite element problems of this formulation. In conclusion, this formulation is found to be useful as a motion study for sample problems of all scales for which the acoustic motion is to be approximated.

BIOGRAPHICAL SKETCH

Mija Hubler was born in 1985 in Munich. Three years later she and her family moved to Urbana, IL. Mija Hubler entered the University of Illinois at Urbana-Champaign in 2002. In 2006 she graduated with honors, completing her Bachelor of Civil Engineering. Beginning that same year at Cornell University, she worked to complete her Masters of Civil Engineering. She plans to earn her Ph.D. in the future.

This document is dedicated to my family and friends.

ACKNOWLEDGEMENTS

This work would not have been possible without the support and encouragement of Prof. Christopher Earls, under whose supervision I chose this topic and wrote the thesis. Prof. Wilkins Aquino also offered many critical suggestions and advice. I also thank my family, on whose constant encouragement and love I have relied throughout all my studies.

TABLE OF CONTENTS

Biographical Sketch	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vi
List of Figures	vii
1 Introduction	1
2 Background	2
2.1 Acoustic FSI modeling	2
2.2 Literature Review	2
2.3 Research Objective	4
2.4 Proposed Methods	5
3 FSI system	6
3.1 Acoustic Fluid Formulation	6
3.2 Continuum Element Model	13
3.3 Dynamic Model	18
3.4 Acoustic Model	22
3.5 Model Input and Output	26
4 Analysis	28
4.1 Regarding the Data	28
4.2 Regarding the Formulation	29
4.3 Regarding the Fluid Acoustics	31
5 Discussion	34
5.1 Systematic Studies of Variables	34
5.2 Other matrix Implementations	35
5.3 Concerns Regarding Verification	37
6 Limitations and Outlook	38
6.1 Acoustic Behavior Assumptions	38
6.2 Approximations in the Formulation	39
6.3 Computational Issues	39
6.4 Inverse Solution Techniques	40
7 Conclusions	41
A Appendix	43
Bibliography	103

LIST OF FIGURES

3.1	A cartoon representing the fluid structure system under consideration.	7
3.2	Steel cantilever model.	17
3.3	The vertical displacement data of the continuum cantilever model.	18
3.4	The time integration algorithm.	20
3.5	The solid model used for dynamic analysis.	21
3.6	Displacement of a node at the center of the solid model in the direction of loading.	22
3.7	Acoustic pressure data at the center of a cube of acoustic fluid elements.	24
3.8	The fluid structure interaction model with a steel cube undergoing plane wave excitation surrounded by acoustic water elements. . . .	25
3.9	The fluid structure model's interface displacement data in the direction of loading, compared with ADINA results.	26
4.1	The frequency content of the fluid structure interaction model displacement data.	30
4.2	Displacement data at interface of interaction model compared to ADINA results for different boundary conditions.	31
4.3	A convergence study of the solid element model with respect to the l_2 norm of displacement data from the loaded surface.	32
5.1	A sensitivity study of the acoustic fluid structure interaction model interface displacement in response to changes of the steel's Young's modulus.	35
5.2	A sensitivity study of the acoustic fluid structure interaction model interface displacement in response to changes of the steel's density.	36

CHAPTER 1

INTRODUCTION

The problem of computing the behavior of compressible fluids in response to structural vibrations has a range of applications. The methods presented here apply the concepts introduced by Zienkiewicz [29], which allow for a direct coupling of such fluid and structure systems to be solved in a finite element framework. This method is particularly suitable for applications to long slender systems such as the oscillations in rocket fuel systems, earthquake motions applied to water retaining structures [8], shipbuilding considerations, and electro magnetic vibration situations [19]. With these applications in mind the details of the mechanical theory and computational methods are investigated in an effort to provide insight on the sensitive parameters of such computational acoustic systems. Possible variations in how these methods are applied are also discussed.

CHAPTER 2

BACKGROUND

2.1 Acoustic FSI modeling

Acoustic fluid structure finite element modeling refers to the specific computational approach for solving structural acoustics problems which is presented in this thesis. Structural acoustics problems in general aim to solve for the acoustic pressure field resulting in a fluid and solid system due to mechanical solid excitation or external fluid excitation. Computational structural acoustics problems may be approached from various mathematical solution techniques. The methods presented here illustrate an approach which couples a finite element model of the structure with an approximate finite element model of the surrounding fluid. The finite element framework is applied to the joined system as one mesh for both fluid and solid elements, with an element-by-element variation in formulation of the material properties to approximate the corresponding behavior.

2.2 Literature Review

In the past 40 years, a number of techniques have been developed to computationally solve acoustic fluid structure interaction systems. Methods have been well developed for solid phase interactions with incompressible fluid media using finite element techniques for both material domains [30] [31]. Yet, more diverse approaches have been developed in order to deal with compressible fluids. The structure is typically represented with elastic finite elements which allow for great geometrical complexity in the model. Techniques for modeling the behavior of the

fluid vary in the computational methods applied and the selection of the primary unknown which is being solved for. Computational methods for the fluid phase have been developed using complex response functions [8], boundary elements [7] [27] [15], finite elements [11] [24], infinite elements [21], analytical approaches [18] [20], and T-matrix methods [26]. The most popular of these methods to this day is the treatment of the fluid phase through boundary elements, yet this does not result in dynamic fluid data throughout the whole fluid region. The combined finite element/analytical method requires prior knowledge about the observed system before analysis. The finite element computation of the fluid dynamics resolves these issues but introduces meshing difficulties. Variations in the primary unknown selected for solution of the fluid include; pressure [32] [9], fluid particle displacement [16] [19], displacement potential [22], and velocity potential [11] [12]. While pressure and displacement formulations provide the most direct solution data, the displacement potential and velocity potential usually allow for symmetric matrices to simplify the required numerical methods [13]. Everstine suggests an analogy between the equations of elasticity and the wave equation, in [10], which is based on methods presented by Zienkiewicz, which allows for the use of finite elements in both the fluid and solid domains with very little change in the element formulation between the two phases.

More recently, acoustic finite element modeling has been used in soft tissue modeling [23] [6] and sea mine detection [17]. Thompson states in [25] that there are four major challenges still present in the field of acoustic modeling with finite elements. The first is the effective treatment of unbounded domains. Current options include using local and nonlocal absorbing boundary conditions, infinite elements, and absorbing layers. Another challenge is present in dealing with numerical dispersion of short waves. This is of particular importance in dealing with

the standard Galerkin method which in cases of short waves produces problems in accurately resolving oscillations at high frequencies. This difficulty of obtaining accuracy at high wave numbers is currently listed as one of the most challenging problems in scientific computation [28]. The last two challenges involve efficient solution techniques for non-Hermitian matrices and *a posteriori* error estimates for the Helmholtz operator for adaptive meshes. In light of these challenges, there is still discussion on error, stability, and discretization methods. Thus, the current analysis of the method details plays a role in furthering acoustic finite element modeling efforts to approach the posed challenges.

2.3 Research Objective

Using the analogy between the equations of elasticity and the wave equation has many computational and methodological advantages. The complete fluid and solid system can be modeled with one finite element mesh which relieves complicated meshing issues. The system as a whole can be expressed in the form of a single banded matrix. Since the fluid element formulation is based on the elasticity equation form, the fluid component can theoretically be added as an added capability to an existing elastic finite element solver. Everstine presents the method using velocity potential as the primary fluid variable in order to create a symmetric matrix system [10]. This technique requires recasting the fluid pressure variable which is the direct product of the finite element system for the fluid in order to perform computations. Consequently, the results also need to be cast back to their original pressure form to make sense of the data. The current objective is to develop this computational method of a combined solid and fluid finite element meshed model which instead uses pressure as the primary unknown and is built on an independent

elastic solid finite element solver.

2.4 Proposed Methods

The Zienkiewicz approximations are applied in a C language code. This method involves the development of three separate, but related, programs. The first program solves a purely solid system subjected to static loads. It was based off of a finite element code provided by Prof. Earls known as BEN. The new finite element solid code is written in a way which would allow for the addition of fluid elements without restructuring. Thus, mimicking the situation of beginning with a commercial code and applying an acoustic fluid structure interaction interface. This program is then developed further into a dynamic solid system solver in order to incorporate inertial effects. Time integration and mass matrix computation options are set to allow for optimizing with the addition of acoustic elements in the next stage of the code. The final version of the code includes the dynamic elastic solid elements and adds the compressible fluid elements with a transformation between the two element types. All stages of the code are verified analytically and the interaction is compared to similar models built in the commercial software, ADINA. For the fluid and solid interaction code the systems are kept under 2000 degrees of freedom and rely on an optimized Lapack solver in order to run in a few minutes on a dual-core 2.6 GHz AMD Opturon Linux workstation.

CHAPTER 3

FSI SYSTEM

3.1 Acoustic Fluid Formulation

The formulation for the interaction of a given elastic structure and an acoustic fluid for finite elements requires a number of assumptions. The structural behavior follows that of the theory of elasticity, incorporating general approximate engineering theories. The fluid is considered a compressible, non-viscous, non-flowing medium. Small motions of the fluid are expected to correspond to small variations in pressure at the corresponding location in the fluid. It is additionally assumed that all of the given fluid behaves according to the wave equation. Starting with the wave equation, we can take a closer look at the theoretical foundation of the acoustic finite element interaction.

Referring to figure 3.1, let us consider an elastic structure with an outward normal vector \tilde{n} , defined on every point of its boundary, Γ . The elastic structure's deformations and outward normals are all defined in reference to a local Cartesian coordinate system consisting of the vectors \tilde{x} , \tilde{y} , and \tilde{z} . The surrounding fluid is also defined by a boundary, Γ_f and fluid properties. Newton's law, $F = ma$ may be applied to describe the state of the fluid in each coordinate direction.

$$\frac{\partial p}{\partial x} = -\rho\ddot{u}, \quad \frac{\partial p}{\partial y} = -\rho\ddot{v}, \quad \frac{\partial p}{\partial z} = -\rho\ddot{w} \quad (3.1)$$

Where ρ is the fluid mass density, p is the dynamic fluid pressure at a given point on the interface, and \ddot{u} , \ddot{v} , and \ddot{w} , refer to the fluid particle accelerations in the given coordinate directions, respectively. If we combine each of these components

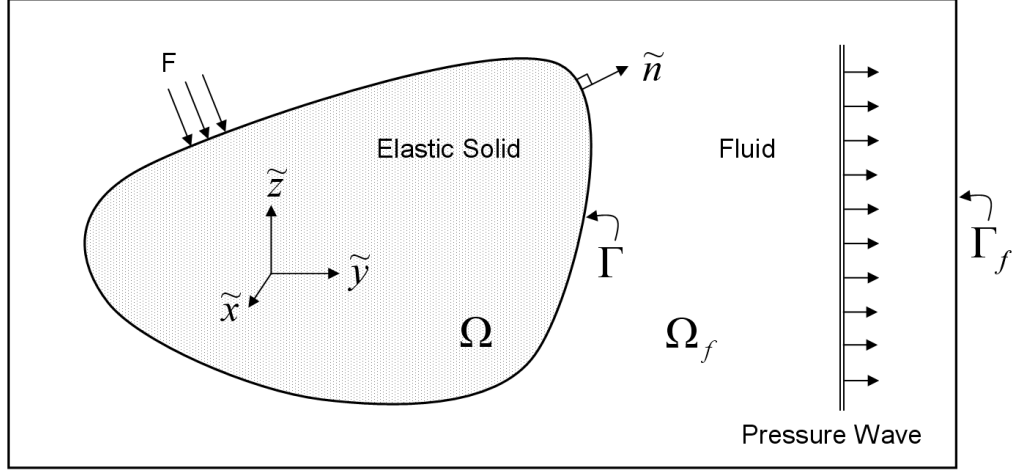


Figure 3.1: A cartoon representing the fluid structure system under consideration.

into a general PDE for the fluid,

$$\nabla^2 p + \rho \ddot{u} = 0. \quad (3.2)$$

To calculate the strong form of this equation we differentiate again with respect to the spatial coordinate.

$$\nabla^2 p + \rho(\ddot{u}_{,x} + \ddot{v}_{,x} + \ddot{w}_{,x}) = 0 \quad (3.3)$$

Here we must assume that since we are dealing with a fluid, the shear modulus is close enough to zero, that we can neglect it. By applying this assumption and setting the three spatial stresses seen corresponding to the behavior of the elastic

solid equal, we arrive at the following relation using the fluid bulk modulus.

$$\sigma_x = \sigma_y = \sigma_z = p = B(\epsilon_x + \epsilon_y + \epsilon_z) \quad (3.4)$$

According to elastic theory, we know that strains are related to displacements as $\epsilon_{,i} = u_{,i}$. Incorporating these relations into our differential equation allows us to rewrite it as,

$$\nabla^2 p + \rho \frac{d^2}{dt^2}(\epsilon_{,x} + \epsilon_{,y} + \epsilon_{,z}) = 0 \quad (3.5)$$

$$\nabla^2 p + \rho \frac{d^2}{dt^2} \left(\frac{p}{B} \right) = 0 \quad (3.6)$$

$$\nabla^2 p + \frac{\rho}{B} \ddot{p} = 0 \quad (3.7)$$

And lastly since the speed of sound, $c = \sqrt{\frac{B}{\rho}}$, the desired wave equation PDE is expressed as,

$$\nabla^2 p + \frac{1}{c^2} \ddot{p} = 0 \text{ on } \Omega, \text{ the element domain.} \quad (3.8)$$

Using the Galerkin method, we define a trial pressure field, δp , on Ω_f and Γ_f , the fluid domain. Since this is an arbitrary function we may multiply our strong form equation with it and maintain validity. Integrating over the whole domain,

$$\int_{\Omega_f} \delta p \nabla^2 p d\Omega_f - \int_{\Omega_f} \delta p \frac{1}{c^2} \ddot{p} d\Omega_f = 0. \quad (3.9)$$

Applying the divergence theorem, the weak form can be further simplified to:

$$\int_{\Gamma_f} \delta p (\nabla p \cdot n) d\Gamma_f - \int_{\Omega_f} \nabla(\delta p) \cdot \nabla p d\Omega_f - \int_{\Omega_f} \nabla p \frac{1}{c^2} \ddot{p} d\Omega_f = 0. \quad (3.10)$$

The boundary, Γ_f , can be broken into the essential, natural, and radiation conditions. The constraints for the boundary conditions I worked with for the fluid structure interaction problem are:

$$\text{high frequency free surface} = 0 \quad (3.11)$$

$$\text{accelerating interface} \quad \frac{\delta p}{\delta n} = -\rho \ddot{u}_n \quad (3.12)$$

$$\text{simplest approximate nonlocal radiation model} \quad \frac{\delta p}{\delta n} = -\frac{\dot{p}}{c} \quad (3.13)$$

According to [10], all constraints of the form $\frac{\delta p}{\delta n}$ can be prescribed as loads at the affected nodes. As a result the surface integral over each element is zero and the weak form becomes:

$$\int_{\Omega_f} \nabla(\delta p) \cdot \nabla p d\Omega_f - \int_{\Omega_f} \nabla p \frac{1}{c^2} \ddot{p} d\Omega_f = 0 \quad (3.14)$$

This equation may be discretized over h elements using the following scheme.

$$p = \sum_h N_i(x) p_i(t) \quad (3.15)$$

Substituting these expressions into the weak form gives,

$$[Q] \{\ddot{p}\} + [H] \{p\} = \{0\} \quad (3.16)$$

where,

$$[Q] = q_{ij} = \frac{1}{c^2} \int_{\Omega} N_i N_j d\Omega_f \quad (3.17)$$

and

$$[H] = h_{ij} = \int \left[\frac{\delta N_i}{\delta x} \frac{\delta N_i}{\delta x} + \frac{\delta N_i}{\delta y} \frac{\delta N_i}{\delta y} + \frac{\delta N_i}{\delta z} \frac{\delta N_i}{\delta z} \right] d\Omega_f = 0. \quad (3.18)$$

This is the typical formulation of the fluid finite elements. The formulation can alternately be derived from elastic finite elements by observing the parallels between the wave equation and the equations of elasticity. For example, the x-component of the Navier equations of elasticity is

$$\left(\frac{\lambda + 2\mu}{\mu} \right) u_{,xx} + u_{,yy} + u_{,zz} + \left(\frac{\lambda + 2\mu}{\mu} \right) (v_{,xy} + w_{,xz}) + \frac{1}{\mu} f_x = \frac{\rho}{\mu} \ddot{u} \quad (3.19)$$

where λ and μ are the Lamé constants and f_x is the x-component of the body force per unit volume. This equation can be written in the general form

$$\nabla^2 \phi + g = a \ddot{\phi} + b \dot{\phi} \quad (3.20)$$

where ϕ is an unknown scalar function of position and time, which in our case in comparing to the elastic formulation is the displacement u . The following relations must also be set in order for this form to hold true:

$$\frac{\lambda + 2\mu}{\mu} = 1, \quad v = w = 0, \quad \frac{\rho}{\mu} = a, \quad \frac{1}{\mu} f_x = g - b\dot{\phi}. \quad (3.21)$$

If we select a shear modulus of a particular element, μ_e . The density and the first Lamé constant can be written in terms of this selected modulus.

$$\lambda_e = -\mu_e \quad (3.22)$$

$$\rho_e = \mu_e a \quad (3.23)$$

The subscript e is included to emphasize the fact that these material properties are merely values assigned to the elements in an effort to approximate the fluid behavior. We can now rewrite the body force as

$$f_x = \mu_e (g - b\dot{\phi}) \quad (3.24)$$

This force is most directly applied as a total force, F_x , acting on each finite element node corresponding to the unit volume of the force. The total force is thus,

$$F_x = \mu_e g V - (\mu_e b V) \dot{\phi} \quad (3.25)$$

where V is the associated element volume. According to the definitions of three-dimensional elasticity, which we are relying on for this formulation of the fluid, the Lamé constants are given as

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \quad (3.26)$$

$$\mu = \frac{E}{2(1+\nu)} \quad (3.27)$$

$$E = \frac{\mu(3\lambda + 2\mu)}{\lambda + \mu} \quad (3.28)$$

$$\nu = \frac{\lambda}{2(\lambda + \mu)} \quad (3.29)$$

where E and ν are the Young's modulus and Poisson's ratio, respectively. In order to satisfy both these definitions and the relations according to the selected element shear modulus in the Navier equation, we must have infinite values for E and ν . This is not feasible for computational purposes. As a result, this variation on the general equation for the fluid finite elements can be solved with elasticity finite elements if the three dimensional region is modeled with three dimensional solid finite elements having,

$$E_e = \alpha \mu_e \quad (3.30)$$

$$\nu_e = \alpha/2 \quad (3.31)$$

$$\rho_e = \mu_e a \quad (3.32)$$

with $(\alpha \gg 1)$. The results are independent of the value selected for μ_e , and thus it is convenient to set this value to 1.

The total pressure in the fluid-solid system, p , can be split into the scattered and incident components.

$$p = p_s + p_i \quad (3.33)$$

The scattered pressure, p_s , will serve as our primary unknown since we already know the applied incident pressure, p_i . Using this new pressure definition, the finite element model formulated for the fluid domain can now be rewritten to incorporate a radiation boundary condition and solid interface.

$$[Q] \{\ddot{p}_s\} + [C] \{\dot{p}_s\} + [H] \{p_s\} = \{F^{(p)}\} \quad (3.34)$$

$[Q]$ and $[H]$ are as defined previously in equation 3.17 and 3.18, while $[C]$ is the “damping” matrix due to the radiation boundary condition and $[F]$ is the “loading” applied to the fluid. A closer look at the boundary conditions is required to compute $[C]$ and $[F]$.

The general form of the boundary conditions is given as

$$a_1 \frac{\delta \phi}{\delta n} + a_2 \phi + a_3 \dot{\phi} + a_4 \ddot{\phi} + a_5 = 0. \quad (3.35)$$

Here, ϕ is an unknown function of space and time. We can use this general form with u as our unknown variable field. At the interface it is known that $\frac{\delta u}{\delta n} = \nabla u \cdot n = u_{,x} n_x + u_{,y} n_y + u_{,z} n_z = (\sigma_{xx} n_x + \sigma_{xy} n_y + \sigma_{xz} n_z) / \mu_e$. If $T_x^{(u)}$ is the x-component of the vector acting on a surface with normal \tilde{n} , $\frac{\delta u}{\delta n} = \frac{T_x^{(u)}}{\mu_e}$.

For a discretized surface, $T_x^{(u)} = \frac{F_x}{A}$, where A refers to the lumped area at each node. Thus, $\frac{\delta u}{\delta n} = \frac{F_x}{\mu_e A}$. Substituting this into the general form of the boundary condition allows for the solution of a “load” which can be applied to each boundary point that effectively enforces the boundary condition.

$$F_x = -\frac{\mu_e A}{a_1} (a_2 u + a_3 \dot{u} + a_4 \ddot{u} + a_5) \quad a_1 \neq 0 \quad (3.36)$$

Let us consider the radiation boundary condition, for which $\frac{\delta p_s}{\delta n} = -\frac{\dot{p}_s}{c}$. This condition only contains a second derivative term corresponding to a_4 in equation 3.36. With the choice of $\mu_e = 1$, we can solve for the specific force in this case as, $F_x = -\frac{A}{c} \dot{p}_s$.

Applying similar reasoning at the interface, $F_x = -A\rho\ddot{u}_{ns}$. The interface boundary condition applied force can be written in terms of the total normal displacement, u_n , which is the most convenient unknown to be used in the solid domain. Using the relation between the scattered and total pressures,

$$\frac{\delta p_s}{\delta n} = \rho (\ddot{u}_{ni} - \ddot{u}_n) \quad (3.37)$$

where the \ddot{u}_{ni} is the second derivative of the normal displacement due to the incident wave, while \ddot{u}_n is the total outward normal component. As a result,

$$F_x = -A\rho (\ddot{u}_{ni} - \ddot{u}_n) \quad (3.38)$$

$$[Q] \{\ddot{p}_s\} + [C] \{\dot{p}_s\} + [H] \{p_s\} - \rho (GA)^T \ddot{u} = -\rho A \ddot{u}_{ni} \quad (3.39)$$

3.2 Continuum Element Model

The first development step of the acoustic finite element code focused solely on the continuum finite element. The continuum element was formulated as an arbitrary, three-dimensional elastic structural element. At this stage the element could be subjected to internal and external time-dependent loads. The loading was taken to be static and the modeled behavior linear. The deformation motion of an elastic continuum is governed by Newton's second law.

$$\rho \ddot{u} = \nabla \sigma + \rho b \quad (3.40)$$

Due to the static condition, we can drop the first term which represents the acceleration effects of the medium. In order to solve our system, we will need to incorporate additional information regarding the material properties of the elastic material. The equation, as is, describes only three scalar components for each coordinate direction, but there are six unknown distinct entries in our symmetric stress tensor. If we assume isotropic, elastic material properties we can make use of the relation between stress and strain through the young's modulus or Hooke's law. Incorporating these into the general PDE gives,

$$(\lambda + \mu) \nabla (\nabla \cdot u) + \mu \nabla^2 u = -\rho b \quad (3.41)$$

where ∇ and μ are Lamé's elasticity constants and E is Young's modulus which may vary in space throughout the material. We can apply a Galerkin method to the previously developed Navier equations for isotropic linear elasticity. The displacement field, u , is spatially approximated by

$$\tilde{u} = \sum_{j=1}^n u_j N_j(x_1, \dots, x_d) \quad (3.42)$$

where N_i are the appropriate finite element basis functions. Replacing the continuous displacement field in the previous equations with this approximation produces

a residual in the equation. This residual, multiplied by a weighting function, must vanish as the solution is approached. The Galerkin method dictates that the shape functions are employed as these linearly independent weighting functions. Thus by integrating by parts we arrive at a weak form of our PDE,

$$\int_{\Omega} \sigma \nabla N d\Omega = \int_{\partial\Omega} N \sigma n d\Gamma \quad (3.43)$$

n is the surface normal on the element boundary Γ . Integration over a general brick element is performed in the parent element according to the following shape functions, and then transformed to the deformed element by utilizing the Jacobian matrix properties.

$$N_I(\tilde{\xi}) = N_I(\xi, \eta, \zeta) = \frac{1}{8} (1 + \xi_I \xi) (1 + \eta_I \eta) (1 + \zeta_I \zeta), \quad I = 1, \dots, 8 \quad (3.44)$$

where I refers to the eight element nodes and ξ_I , η_I , and ζ_I correspond to their coordinates in the respective coordinate directions. As a result, coordinate interpolation is done in terms of the shape functions as,

$$x = \sum_{I=1}^8 N_I(\tilde{\xi}) x_i. \quad (3.45)$$

The Jacobian matrix, $[J]$, relates the natural coordinate derivatives to the local coordinate derivatives.

$$\frac{\partial}{\partial \tilde{\xi}} = [J] \frac{\partial}{\partial \tilde{x}} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \quad (3.46)$$

By making use of the Jacobian matrix, the spatial derivatives of the displacements at any point in the element can be interpolated and then calculated from the element node displacements using the shape functions. Considering the x-direction

this would be,

$$\begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \\ \frac{\partial u}{\partial z} \end{bmatrix} = \frac{1}{8} [J]_{ij}^{-1} \begin{bmatrix} \frac{\partial u}{\partial \xi} \\ \frac{\partial u}{\partial \eta} \\ \frac{\partial u}{\partial \zeta} \end{bmatrix} = \frac{1}{8} [J]_{ij}^{-1} [B] \begin{bmatrix} u_1 \\ v_1 \\ w_1 \\ u_2 \\ v_2 \\ w_2 \\ \vdots \end{bmatrix}. \quad (3.47)$$

$[J]_{ij}^{-1}$ is the inverse Jacobian matrix and $[B]$ is the strain-displacement transformation matrix defined as follows.

$$\tilde{\xi} = [B] \tilde{u} \quad (3.48)$$

Now we can apply these approximations to our weak form integrals.

$$\sum_{j=1}^n \int_{\Omega} [B]^T [D] [B] d\Omega \{u\} = \int_{\Omega} [B]^T [D] \{\tau\} d\Omega \quad (3.49)$$

The system force vector can be written as $\{F\} = [B]^T [D] [B] \det [J]$. The system stiffness matrix is related to this force through a set of weight functions, α .

$$[K] = \sum \alpha [F] \quad (3.50)$$

According to [3] these are always 1 for our case of a three-dimensional brick element. The current problem is assumed to be static, thus the forces are all time independent and the system is only displacement dependent. At any solution time the system can be described as,

$$[K] \tilde{u} = R_B + R_S - R_L + R_C \quad (3.51)$$

where R_B refers to the load vector due to body forces, R_S refers to the load vector due to surface forces, and R_L due to the initial stresses and R_C due to concentrated loads. This system equation allows for direct solution by division.

To begin implementation of the method, the code stores the element connectivities and location from the input file. Based on the methods described by Bathe and Wilson [4], an integer vector, named jcode, is then written which carries the degree of freedom information separately. All solid nodes are specified by three degrees of freedom for each direction of motion the that node. They are zeroed out in case of a fixed boundary, and then numbered consecutively according to the element numbering scheme. The joint code is then used to write the mcode data vector which stores the same information but in order and according to the arrangement of the element connectivities. This data storage method allows for the determination of the active degrees of freedom in the specified order. All the matrices required for solution of the system are arranged according to this numbering of active degrees of freedom. In order to calculate the stiffness matrix, mass matrix, and damping matrix the matrix components are first calculated at each node for each element using local coordinates in conjunction with a general Jacobian matrix calculation function. Using the mcode data vector these element matrix values are then assigned to the correct global degree of freedom locations in the general system matrix. At each load step the external applied force is updated and multiplied with the global matrix to solve for the system displacements in a solve function. Finally the displacements are written to an output file for each active degree of freedom.

Verification of the this initial static elastic code was performed with a cantilever model test. The model is shown in figure 3.2 and consisted of 576 solid elements. One surface was fixed at the end of the cantilever and loading was applied perpendicular to this fixed surface at a node at the other end in the vertical direction. The materials properties were chosen to approximate steel material with $E = 29,000psi$, $\nu = 0.3$, and $\rho = 0.02lb/in^3$. As shown in figure 3.2, the results

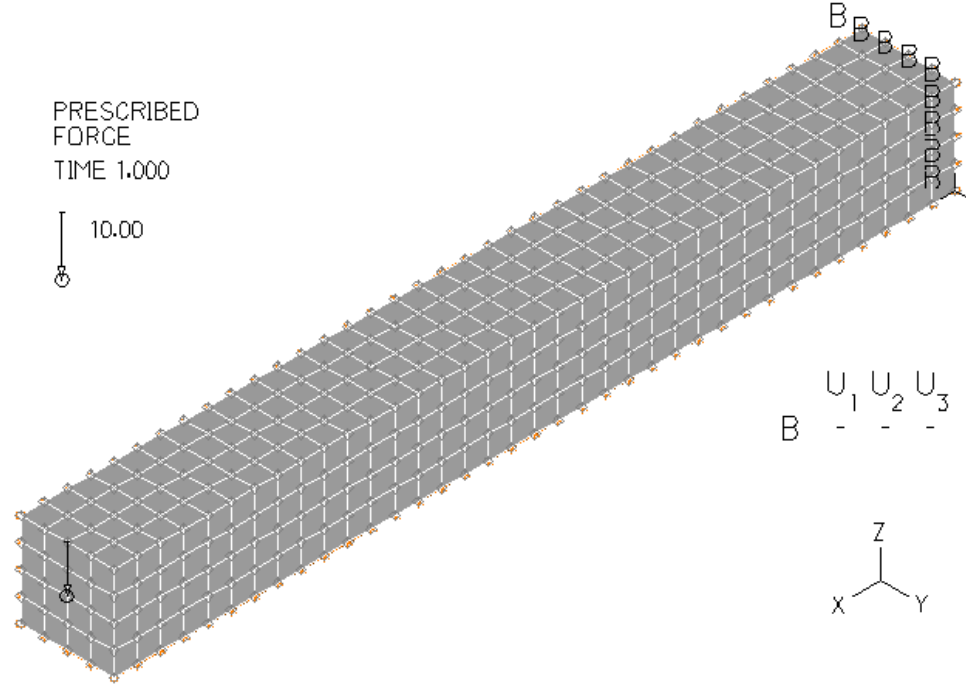


Figure 3.2: Steel cantilever model.

have a slight discrepancy with the analytical solution for a deflecting cantilever given by,

$$w(x) = \frac{-Px^2(3L - x)}{6EI} \quad (3.52)$$

where the deflection, w , is in terms of the position x along the cantilever, and depends on the load, P , the length, L , and the Young's modulus and axial moment of inertia, E and I .

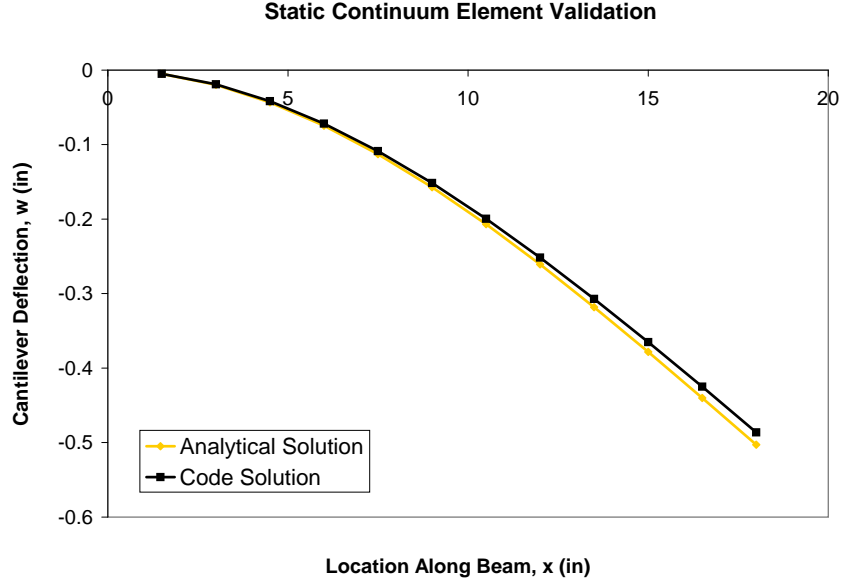


Figure 3.3: The vertical displacement data of the continuum cantilever model.

3.3 Dynamic Model

In order to add the dynamic effects to the continuum model the first term of equation 3.40 is now required in order to incorporate acceleration effects into the system's motion. The central difference method is the best to use in this case because the problem requires second derivatives of displacement. The central difference method offers a convenient method for approximating these derivatives while efficiently solving the system of equations using the following approximations,

$$a^t = \frac{1}{\Delta t^2} (u^{t-\Delta t} - 2u^t + u^{t+\Delta t}) \quad (3.53)$$

$$v^t = \frac{1}{2\Delta t} (-u^{t-\Delta t} + u^{t+\Delta t}) \quad (3.54)$$

The true strength of the central difference method lies in being able to avoid matrix inversion in the case of diagonal matrices. Although it is not feasible later in the acoustic implementation of this problem, this strength will be used to our advantage at this stage. If we substitute the expansions for acceleration and velocity into the general matrix equation for the solid,

$$[M] \{a\}^t + [C] \{v\}^t + [K] \{u\}^t = \{R\} \quad (3.55)$$

we obtain the required central difference equation to solve with unknowns on the left-hand side, and known quantities on the right.

$$\left(\frac{1}{\Delta t^2} M + \frac{1}{2\Delta t} C \right) u^{t+\Delta t} = R^t - \left(K - \frac{2}{\Delta t^2} M \right) u^t - \left(\frac{1}{\Delta t^2} M - \frac{1}{2\Delta t} C \right) u^{t-\Delta t}. \quad (3.56)$$

To start the algorithm, a starting condition must be specified as:

$$u^{-\Delta t} = u^0 - \Delta t v^0 + \frac{\Delta t^2}{2} a^0 \quad (3.57)$$

The algorithm for time integration is presented in figure 3.4 and is based on one presented in [5].

If the mass and damping matrices are both diagonal, the system of equations can be solved without factorization i.e.

$$u_i^{t+\Delta t} = \hat{R}_i^t \left(\frac{\Delta t^2}{m_{ii}} \right) + \hat{R}_i^t \left(\frac{2\Delta t}{c_{ii}} \right) \quad (3.58)$$

In order to create a diagonal mass matrix a consistent mass matrix is calculated using the previously developed formulation. Then all terms along each row are summed and placed on the diagonal. This simplification is particularly applicable for dynamic systems since the inertial forces are acting at each degree of freedom. To ensure a diagonal damping matrix, mass-proportional damping must be used such that.

$$[C] = \alpha [M] \quad (3.59)$$

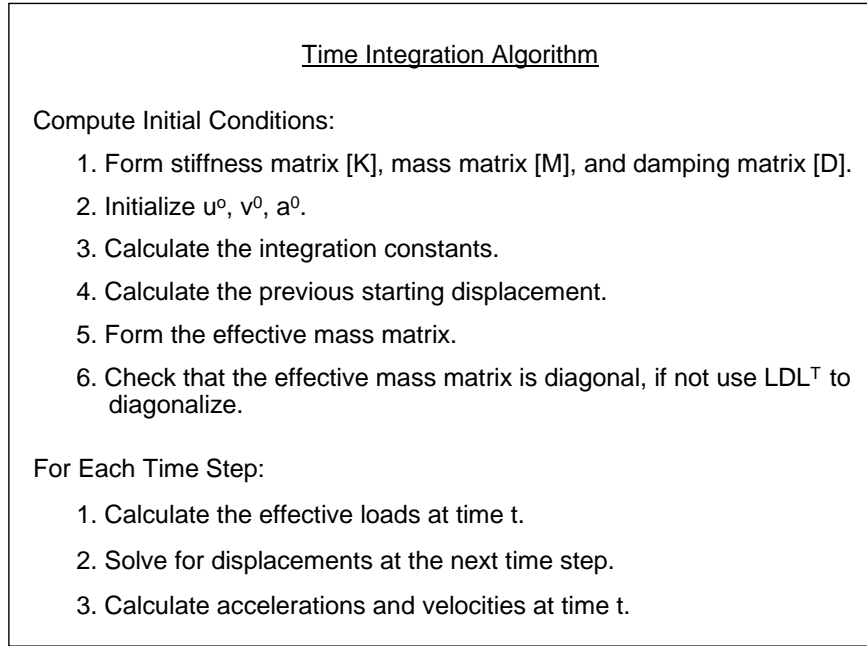


Figure 3.4: The time integration algorithm.

α is the proportional damping factor which for large ships made with A36 steel would be around 0.02.

After setting up the system matrices, a time step loop is entered in the dynamic code. At each solution time step the internal force vectors are calculated and the current loading is determined. From these force vectors the effective load vector is computed. The effective load vector is then divided by the effective mass matrix to solve for the displacement at the current time step. With the accelerations and velocities at the next time step forecast using the central difference method, the current time step solution is then complete. In an effort to gauge the acceptability of the dynamic code solution, a sample problem was solved to compare the resulting displacements with known traveling wave solutions. A cube of continuum elements

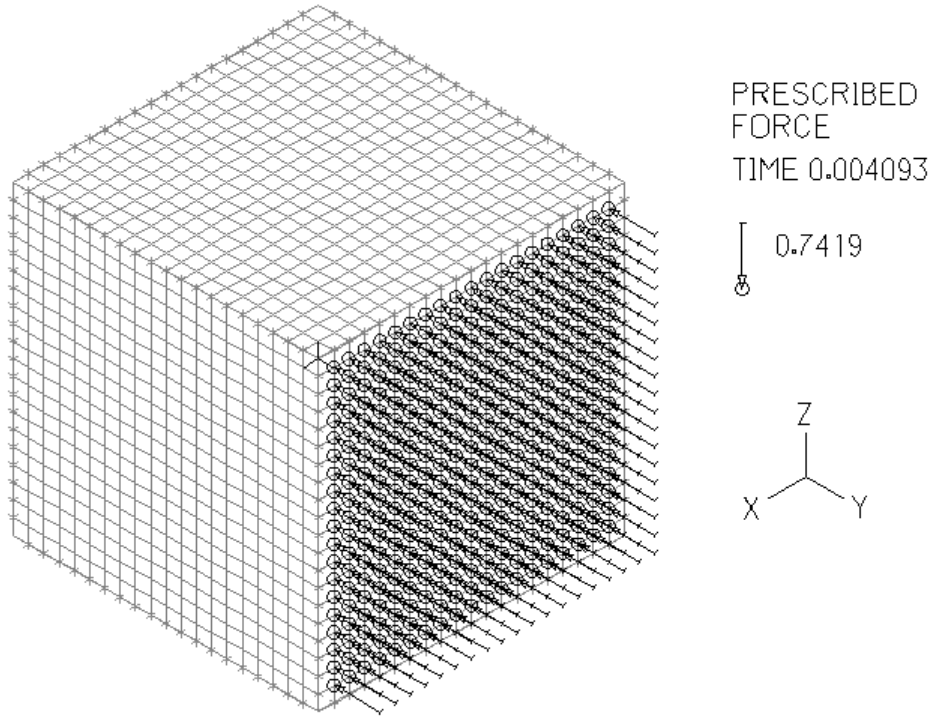


Figure 3.5: The solid model used for dynamic analysis.

is subjected to a plane wave generated on one of the surfaces as shown in figure 3.5. Dirichlet boundary conditions following the applied waveform are specified along all other boundaries to avoid interaction effects. The plane wave is a harmonic wave with a set frequency of 1000Hz. The sample problem was compared for four different mesh densities, all of which showed similar results. The resulting data is shown in figure 3.6. Here too, there are slight discrepancies.

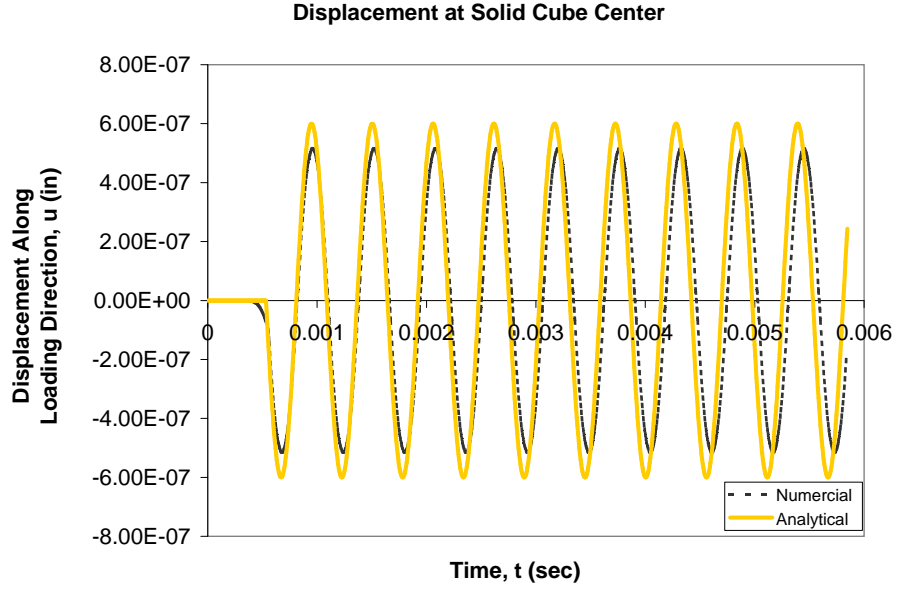


Figure 3.6: Displacement of a node at the center of the solid model in the direction of loading.

3.4 Acoustic Model

Focusing on the acoustic fluid element according to Zienkiewicz's formulation the general finite element formulation for the acoustic fluid region is,

$$[Q] \{\ddot{p}_s\} + [C] \{\dot{p}_s\} + [H] \{p_s\} = \{F^p\}. \quad (3.60)$$

Since the total pressure is given as $p = p_s + p_i$, the incident pressure is known, and the scattered pressure may be solved for. The applied force only consists of the pressures applied to the fluid. The first step is to identify the acoustic element and solid interface normals. This can be accomplished by checking all adjacent element definitions for each prescribed acoustic element and then mapping its

orientation to a parent element. All matrix components listed in the equation above are calculated using the available stiffness, damping, and mass matrix functions for the geometrically similar solid element functions. The assembly functions are called a second time after writing the solid matrix entries, this time using the fluid properties which have been adjusted as dictated by equations 3.30-3.32. The required element data storage vectors are assigned to the value zero in the second and third degree of freedom at each active node. The first degree of freedom is kept active to store pressure data at each unconstrained fluid element node. As a result, the first section of the global system unknowns correspond to the solid displacements and then are followed by the unknown scattered pressures from the fluid domain and on the interface. This leads to the off-diagonal matrix quadrant having to provide a transformation from the three degree of freedom per node data to the single pressure degree of freedom fluid pressure. First, the normal component of the displacement to the interface surface propagates a pressure wave into the fluid. The transformation matrix is of the form $[L] = [G][A]$, where $[G]$ performs the geometric transformation and $[A]$ assigns the interface area to the associated degree of freedom. $[L]$ transforms fluid pressures to solid displacements, and the product of its inverse and the fluid density provide the reverse transformation. Due to the transformation components, the matrix has off-diagonal terms at each of the active interface degrees of freedom. To solve the combined system a decomposition is required, one which would be provided through an LU solver. To improve the runtime of larger models, an optimized LAPACK solver function was incorporated into the existing acoustic code. The LAPACK subroutine, DGESV, solves a matrix system using LU decomposition with partial pivoting and row interchanges of any N by N matrix. Documentation is provided in [2].

In an effort to verify the fluid domain behavior, the same model as shown in

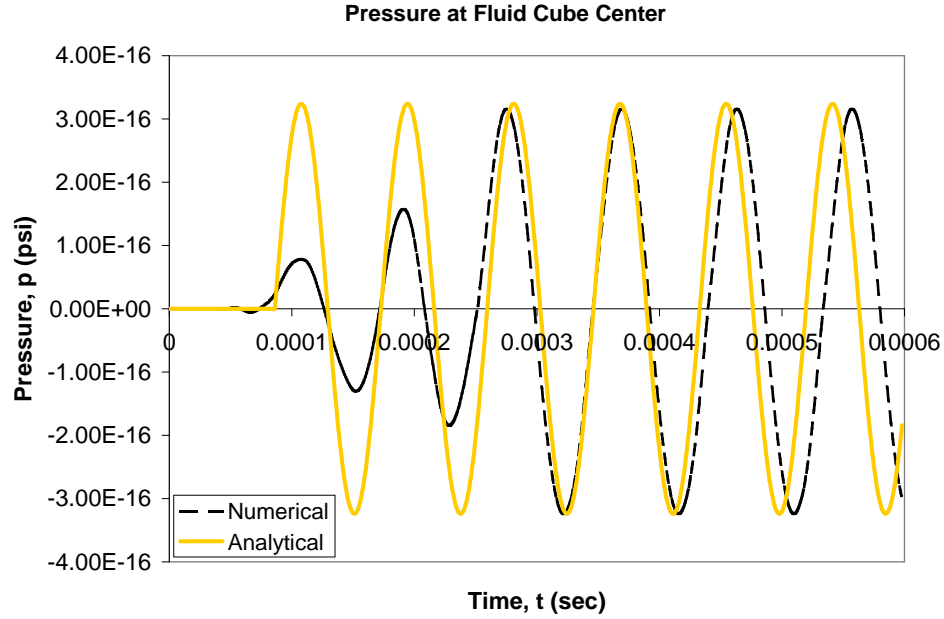


Figure 3.7: Acoustic pressure data at the center of a cube of acoustic fluid elements.

figure 3.5 was studied with acoustic elements throughout the full domain. Pressure data was taken at the center of the cube as the harmonic plane wave was generated at one surface using applied pressure. The numerical results are plotted on results of a damped pressure wave moving through water without boundary condition effects in figure 3.7. There is clearly a large discrepancy between these results. A similar slow starting behavior was observed in ADINA fluid elements, for which I have no explanation.

A sample problem was also studied to look at interaction effects. This model took a smaller cube and placed it in a larger cube fluid volume. The solid domain was meshed with 27 elastic elements and the fluid domain consists of 2170 acoustic fluid elements. The system solves in approximately 1 sec per loading time step

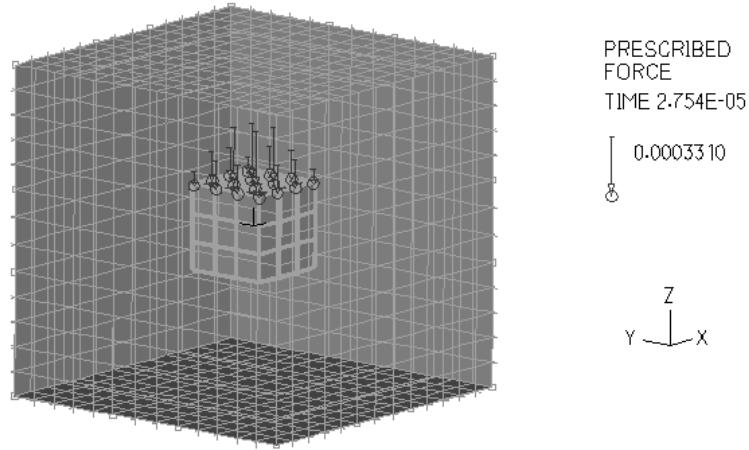


Figure 3.8: The fluid structure interaction model with a steel cube undergoing plane wave excitation surrounded by acoustic water elements.

specified. A tone burst comprising of a modulated sine wave was applied as a plane displacement wave to the cube surface. The cube is fixed at the surface opposite to the applied loading. The fluid boundary was constrained with a high-frequency condition specified as zero pressure immediately as the wave hits the boundary. Damping was set to zero in both the solid and fluid. Material properties were selected to reflect those of steel and water. In order to compare the computational output, a similar model was built in ADINA. The ADINA model uses fluid potential elements which require an implicit calculation scheme and treatment of the boundary conditions by zeroing the velocity potential. The results from each of these models is plotted in figure 3.9. Further discussion of these results and a

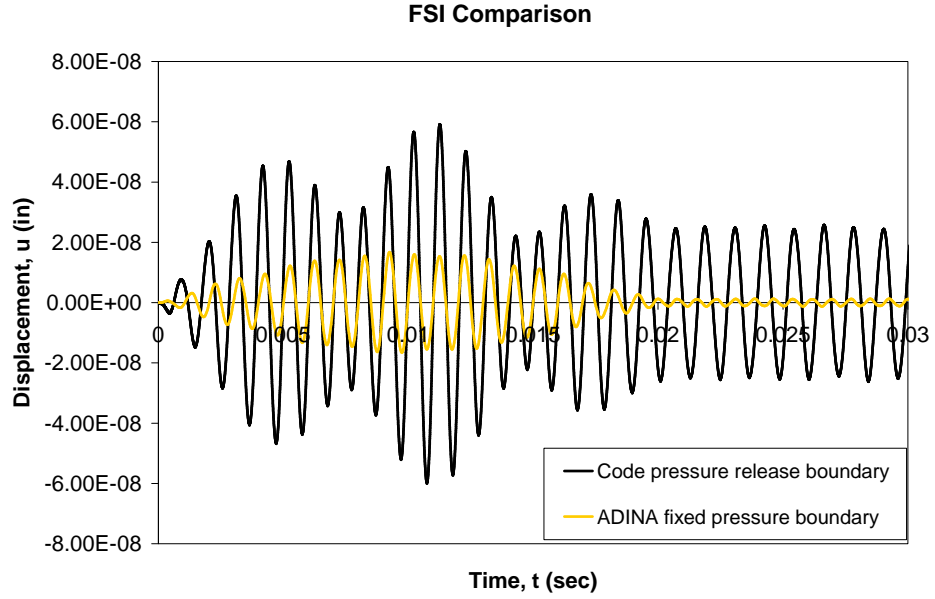


Figure 3.9: The fluid structure model's interface displacement data in the direction of loading, compared with ADINA results.

convergence study are presented in the analysis.

3.5 Model Input and Output

The acoustic model assumes complete knowledge of the geometric properties of the fluid-solid system. It does not require a single solid, or that the fluid surround the solid completely. As a result, a porous or partially submerged structure may also be analyzed. Other than the model geometry, the material properties of all phases, a loading function with given time steps, and mesh connectivities must also be specified. The time function can have multiple stages of loading with varying

time step size, yet the loading time step also controls the integration step and can affect solution stability. The model requires a carefully formatted input file for reading. The number of both solid and fluid elements, as well as the number of interface surfaces is specified to start. After each set of element nodes, a character flag of either “S” or “F” differentiates between solid and fluid properties to be assigned to that element. Other required data includes the domain edge nodes, nodal coordinates, fixed degrees of freedom, loading on nodes, and the loading frequency and time step. Upon running the program, the output file repeats the input geometry and material properties to check for read errors. The displacement data is then output at each time step for all specified result nodes along with the applied load at that time. This data may then be filtered using a Fourier filter code written as a Matlab script. By transforming the displacements through time into the frequency domain, a high-pass or low-pass frequency can be specified. In filtering out this frequency and transforming the data back into the time domain, the filtered signal data may be analyzed to study system harmonics due to the applied loading.

CHAPTER 4

ANALYSIS

4.1 Regarding the Data

In the interaction model, the acoustic model data is presented in the form of solid displacements and fluid pressures at a point on the interface surface. The solid displacements are listed in the direction of loading, which corresponds to the maximum motion observed in the first time step of the system solution. Further interpretation of both the displacement and pressure data at a point on the interface is given by Fourier frequency content analysis of each unknown. Figure 4.1 shows that the interacting fluid response does indeed match that of similar solutions in exception of an observed low frequency beat overtone. This beat in the response could be a possible system response and the solution still falls into the the range of anticipated observations. For example, the tone burst loading function aims to excite a small range of frequencies in the system. This is observed, in addition the following list of response expectations would apply. Firstly, through the addition of the fluid around the solid continuum, the response frequency range is expected to broaden since the fluid elements add more degrees of freedom to the system. An increase in the number of degrees of freedom also increases the number of modes the combined system has available to incorporate into the response. When using realistic material properties of the fluid and solid phase, the speed of sound in the fluid will always be less than the speed of sound in the solid. Across the material interface the spatial wavelength remains unchanged, but the wave prorogation slows to match the speed of sound in the fluid domain. More specific predictions of the combined system response cannot be made without understanding the fluid

boundary condition effects. Since the boundary specifications vary between the current code and ADINA, this point may be considered as a variable in the data comparison. It is certain that due to the fluid, the solid system's response should be damped in amplitude from the added mass of the fluid. Similarly, it is required that the propagating wave across the interface should be slower to correspond to the fluid properties specified in the model definition. Both of these observations can be made in the result data. As seen in figure 4.2, the code pressure release boundary condition produces results which fit between the response of the ADINA free surface solution and fixed pressure boundary. The results suggest that the pressure release provides a slightly more fixed free surface boundary option than ADINA provides with its acoustic elements.

4.2 Regarding the Formulation

The process of developing and applying the acoustic finite element code has led to a number of insights about the formulation by Zienkeiwicz. Observations about the system response may also be developed for the specific model systems selected. As applied to a fully understood and controllable solid system solver, the fluid interaction formulation may indeed be easily integrated into the pre-existing system. All updates to the code may be done by calling the existing integration and matrix population functions from the program main. The same data and element structures can be reused for the fluid portion of the model and interface conditions. Slight improvements did need to be made to the solver to increase its efficiency. This is required since in the combined fluid and structure matrix sparsity is heavily dependent on the overall geometry of the solid elements. To reuse the structural boundary condition assignment process, each node of the fluid boundary had to

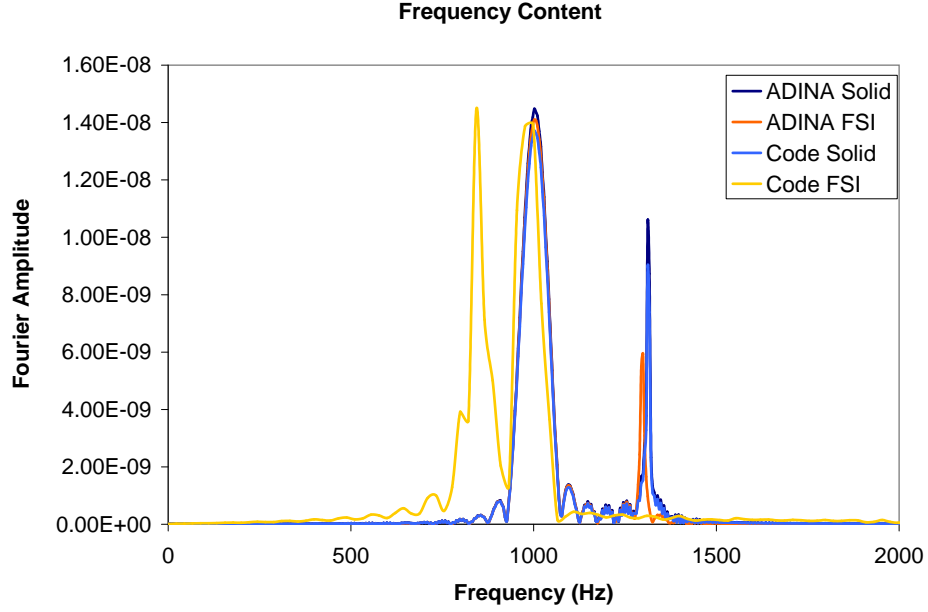


Figure 4.1: The frequency content of the fluid structure interaction model displacement data.

be fixed. This creates inconveniently large input files since the fluid domain is also required to be generously sized to accurately approximate boundary effects using the suggested load analogy method. Mesh generation is simplified since the two model domains are divided into a single mesh. Yet, this creates complications in determining the appropriate degree of freedom from the output data. After removing fixed degrees of freedom and assigning only the active and correct number of degrees of freedom according to the material properties to each node, it is complicated to select the correct unknown from larger systems. To temporarily simplify this issue all loads were applied uniformly according to element area to the models presented here, so that all maximum displacements corresponded to

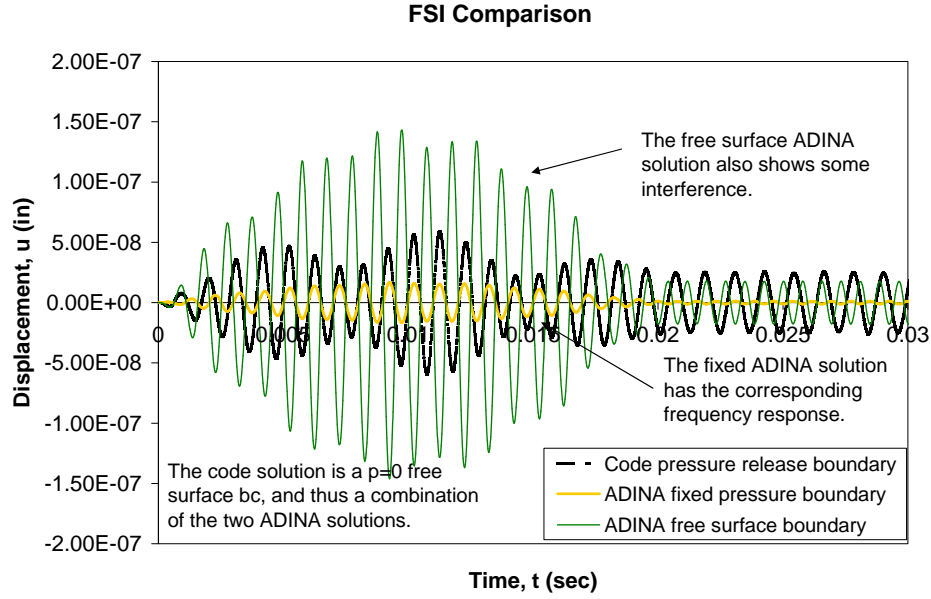


Figure 4.2: Displacement data at interface of interaction model compared to ADINA results for different boundary conditions.

the location of applied load in the first time step. This simplification ensured that the degree of freedom was on the correct surface of the solid. Lastly, the method of applying the boundary conditions in the form of loads has the advantage that interface interactions are very clearly defined in their effects on the structure.

4.3 Regarding the Fluid Acoustics

The dynamic integration of the model introduces sensitivities to the time step size since explicit solution methods are only conditionally stable. Their stability is dependent on the selection of a stable time step for integration. This is referred to

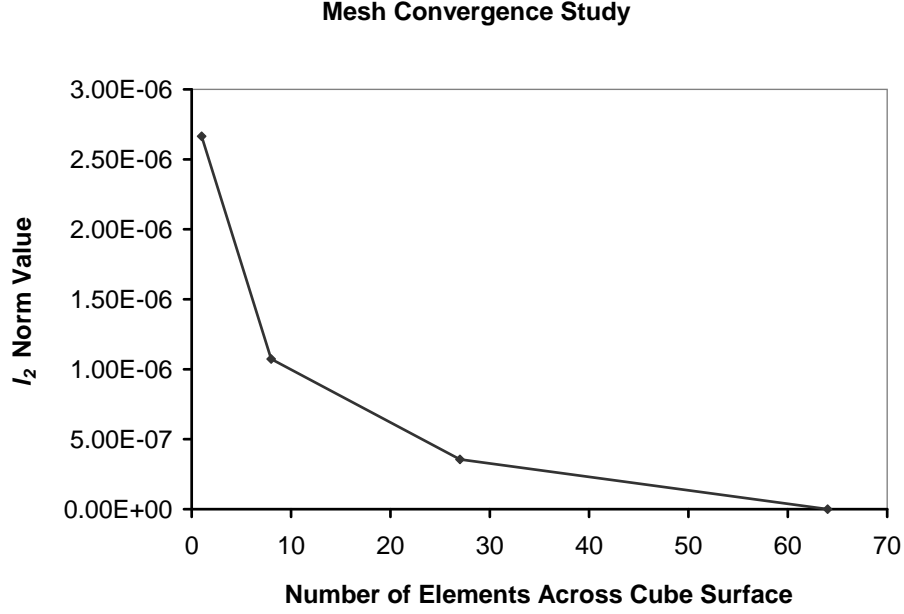


Figure 4.3: A convergence study of the solid element model with respect to the l_2 norm of displacement data from the loaded surface.

as the stability limit and is approximately equal to the time it would take for one elastic wavelength to travel across the smallest element in the mesh. Specifically for the central difference method, the critical time step for any system is calculated as that for a linear undamped system. The linear system calculation is not the same as that for the nonlinear system, but it provides a good indication of where stability will occur in both. The critical time step is calculated as,

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \quad (4.1)$$

where ω_{max} is the maximum eigenfrequency, stemming from the maximum eigenvalue λ_{max} of the generalized eigenvalue problem $Kq = \lambda Mq$, by $\omega^2 = \lambda$.

In order to apply this estimation to the acoustic model, it is necessary to

start with a converged mesh to select the minimum length scale accurately. A convergence study was first performed on the dynamic solid model to determine the required scale without the influence of instabilities the fluid introduces. The mesh converged according to a spatial l_2 norm which approached zero as the element size was decreased. This is plotted in figure 4.3. The converged solid model with a minimum element length of 0.5in was then used in the acoustic fluid interaction model. The fluid element size was matched to the solid element to simplify mesh generation. As a result, the optimal time step could be calculated from this element length.

CHAPTER 5

DISCUSSION

5.1 Systematic Studies of Variables

Due to the implementation of explicit integration, a sensitivity to time step size is expected. The acoustic model showed these sensitivities both with time steps too large and too small. Solutions only converged in a small range around the critical time step. Other than this expected sensitivity, a short study was conducted to observe the sensitivity of the acoustic model in response to changes in the elastic Young's modulus and elastic density. The results in figures 5.1 and 5.2 show that both affect the system response.

The sensitivities were quantified by taking the l_2 norm of the interface displacement data in reference to the median parameter value. The parameter range observed was selected from the observed natural variation of these material properties for steel at various environmental conditions. The study indicates that the steel density has a more linear effect on the system response than the steel's Young's modulus. In addition, in the range observed, the Young's modulus shows a less sensitive response below the median value. If used as solution tool in an effort to determine unknown system parameters in reverse, the solver would be effective in providing data. Yet, for this system more parameters may need to be observed to provide a fuller picture of the response.

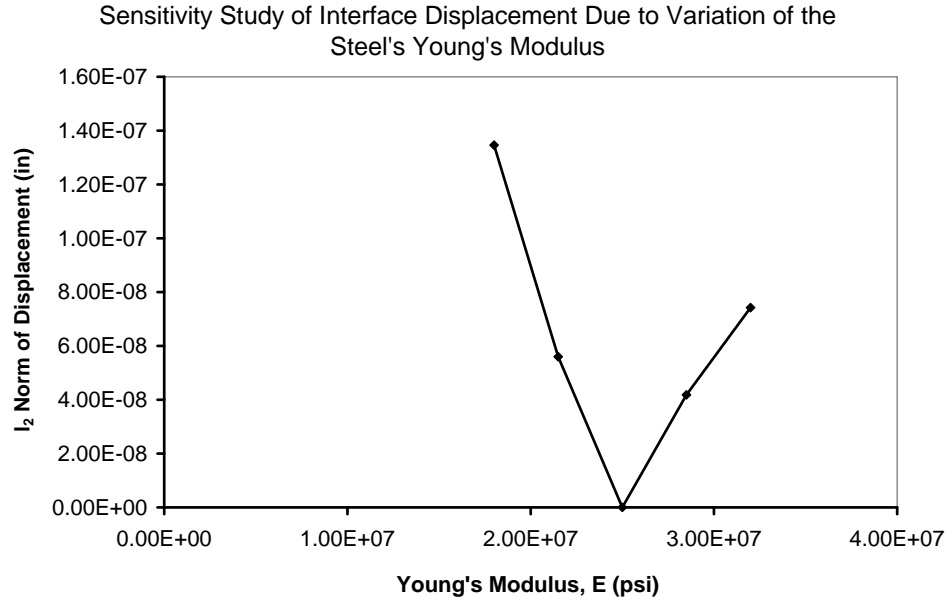


Figure 5.1: A sensitivity study of the acoustic fluid structure interaction model interface displacement in response to changes of the steel's Young's modulus.

5.2 Other matrix Implementations

Everstine does not specify whether the combined mass matrix is calculated in a consistent or diagonal form. The explicit formulation dictates that a lumped mass matrix is best since the inertial effects are applied as lumped body forces, and thus little is gained from the added interpolation in the consistent mass matrix. Although a typical incentive to use the central difference method is that systems may be solved more simply with a lumped mass matrix, this does not apply in our case. The fluid and solid interaction interface creates off-diagonal terms even when the mass matrix is lumped. The condition number of the acoustic model was

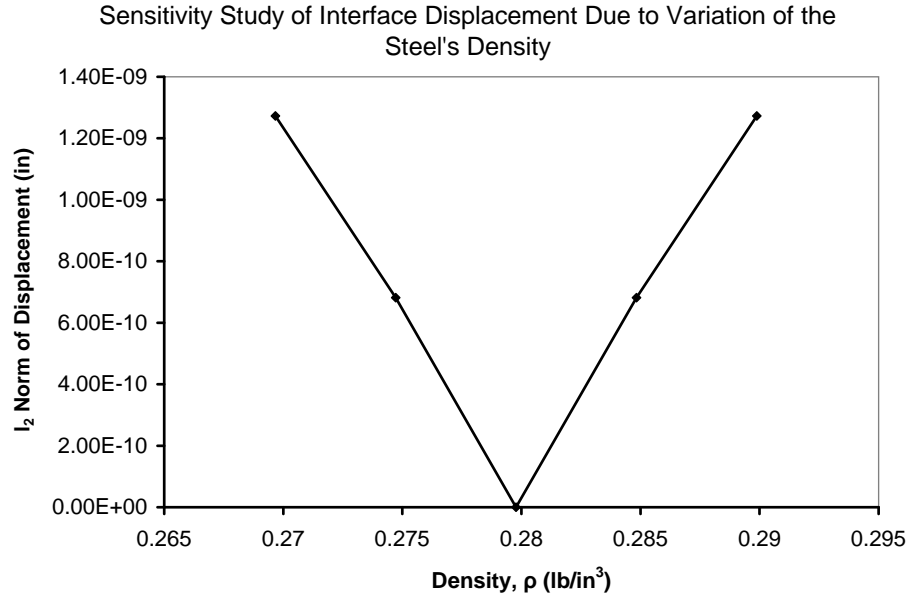


Figure 5.2: A sensitivity study of the acoustic fluid structure interaction model interface displacement in response to changes of the steel's density.

calculated for the system mass matrix. Lumping the consistent matrices resulted in a reduction of $0.5E18$ to $1E9$. The solutions from the consistent mass matrix are not only unstable but their accuracy is lower than the input data values. Very small systems may be solved with a consistent mass matrix for both the fluid and solid components, and have been shown to give the same results as the lumped mass matrix. Therefore the acoustic code is normally run using the lumped mass matrix with the off-diagonal transformation terms. The same matrix layout is applied in cases of structural damping. Mass proportional damping is applied to the lumped mass matrix, resulting in a diagonal structural damping matrix. For boundary condition cases, such as the radiation boundary condition, there are

off-diagonal elements in this matrix as well.

5.3 Concerns Regarding Verification

The fluid structure interaction model solutions were compared to the ADINA software solution since analytical data is only available for problems with cylindrical geometry [14]. The three dimensional brick elements did not allow for such systems. ADINA provides acoustic potential elements that model non-flowing fluid effects and fluid structure interactions [1]. The approximations applied in ADINA may cause concern in comparing the data to that of this acoustic formulation. ADINA approximates mass dynamics incorporating the fixed boundary elements. This theoretical difference should vanish for larger meshes, yet is apparent as an increased mass in dynamic solutions of smaller systems. ADINA only allows for implicit time integration when using the acoustic potential elements in a structural interaction model. Again, for large enough systems with the correct time steps this should also be of little consequence when comparing results. Lastly, ADINA applies boundary conditions to the potential elements in the form of potential interface elements. These elements approximate the effects of a rigid wall, infinite domain, or interface with another fluid. Yet, these elements cannot apply the boundary effect in the same way as the coded formulation since the primary unknown is the fluid potential velocity. As a result differences in the fluid boundary condition will affect results as is seen in the data.

CHAPTER 6

LIMITATIONS AND OUTLOOK

6.1 Acoustic Behavior Assumptions

The assumption of working with a compressible fluid theoretically introduces additional complex analysis phenomena into the system [3]. The methods applied to compressible fluid modeling and incompressible modeling are directly transferable. The most complex issue arising from modeling a compressible fluid is the calculation of shock wave fronts. Although Everstine specifies the formulation we are applying to be designed for compressible fluids, shock waves were not of interest for the acoustic fluid structure interaction model considered. Further development of this code should consider this complex phenomena.

The specified application of boundary conditions have been implemented in similar problems by Everstine. Yet, since exact comparison has not been possible with analytical, experimental, or other computational methods for this aspect of the formulation, a different approach for handling the boundary conditions may be more effective. One method to consider would be an exact non-local technique such as that proposed by Keller in [20]. This technique is optimized for solving the reduced wave equation in an infinite boundary domain. It is non-local, but does not affect the computational efficiency of the solution. Although such an approach would not allow for reflections from fixed boundaries, it would be a practical and effective treatment of elastic objects in large fluid domains.

6.2 Approximations in the Formulation

According to Everstine [13], setbacks to using finite elements for fluid modeling include a need for approximate boundary conditions, problems with requirements on mesh size and extent [19], and difficulty generating the mesh for the fluid. The boundary conditions have indeed posed the greatest challenge for working with Zienkeiwicz’s formulation and finite elements. Yet, mesh generation was simplified by the use of one element and a constant element size. This technique is not optimal since the wave speed is lower in the fluid domain, and thus larger elements would have been adequate. For larger meshes, not only larger elements, but also elements with other geometries would be useful in correctly representing the fluid domain.

6.3 Computational Issues

The converged mesh for the acoustic model presented is relatively course, although it is adequate under convergence criteria. The time step size also requires careful consideration. Both the mesh density and time step are sources of computational instabilities, errors, and increased computation time. Everstine [13] suggests that when working with finite element methods in acoustic problems, it is best to work with structures most suited to the matrices involved. The more diagonal the combined system matrices are, the easier issues of instability are dealt with. Acoustics problems suited for solution with fluid structure interaction finite element modeling are thus large slender structures such as ship hulls, pipes, and vessels. When working with structures of other forms boundary element, infinite element, or non-local methods may be more practical for computation convergence. An additional

concern reported by [16], regarding the formulation, was the generation of spurious modes. This issue arises in time-harmonic problems with zero shear, but was not directly observed in the problems presented here.

6.4 Inverse Solution Techniques

A further application of this type of formulation would involve the development of an inverse solution scheme to make it applicable for structural flaw damage detection. Such a scheme could be implemented by first conducting a thorough sensitivity study of all the model parameters. Once the sensitivities are well understood a fitness function could be developed to correlate changes of a selected parameter to inversely solve a standard acoustic problem. With the incorporation of the developed forward model, an inverse algorithm such as a genetic algorithm, could be used to solve for an approximate solution of the changed parameter. Although this system may not show proved convergence, it should be able to provide the user with a close estimate of the elastic properties of the submerged elastic body.

CHAPTER 7

CONCLUSIONS

The goal of this work was to study and apply Zienkiewicz's formulation for compressible fluid structure interaction acoustics problems in a finite element code. The code was carefully developed using C and was based on Everstine's approach to the formulation. In its finished form the code was used as a tool to study the behavior of a harmonically loaded steel cube submerged in water. This study included model convergence tests and sensitivity studies of the steel density and Young's modulus. All results of the code were compared as closely as possible to a similar model developed using ADINA's potential fluid elements. As a stand alone product of the work, the acoustic finite element solver provided some interesting features. Since it was written in three stages; static elastic, dynamic elastic, and then with added fluid interactions, its development showed how well Zienkiewicz's formulation could be added to an existing software when there are few interaction issues or unknowns in the structural base solver. The resulting acoustic solver can also easily be adapted and added on to incorporate other fluid element formulations. The boundary conditions are designed to work just as accurately for smaller toy problems as well as larger systems. More developed software often uses boundary condition approximations that can have an effect on solutions of low frequency acoustic problems where the boundary plays a critical role in the system response. Lastly, even though the code is less complex in formulation than commercial methods, it makes use of an optimized LaPack solver routine for the matrix inversion, allowing it to be fast enough to tackle larger systems with this approach. The input and output files are read and written in a general manner so that the solver can work with any meshing software to develop models. In conclusion, the acoustic finite element solver based on Zienkiewicz's formulation

is a useful research tool to study fluid and solid motions. Further developments may involve more general boundary condition application options. Other interesting studies could be done with higher frequency shock effects, making use of the compressible fluid capabilities of the formulation.

APPENDIX A

APPENDIX

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <atlas_enum.h>
#include "clapack.h"

#define INPUT "model_def_mesh3fsi.txt" // Map of path to input file
#define OUTPUT "resultsmesh3fsi.txt" // Map of path to output file

/*
Alexander Version 1.0 (April 5, 2009)
Linear finite element analysis for 24-dof brick / continuum element
and acoustic fluid element

Input data:
  units: kip, inch, radian
  enter number of solid elements, number of fluid elements,
    number of solid nodes, number of fluid nodes (in main)
    - ne,nef,nn,nie,nbc
  enter element vertex nodes followed by element type (in struc)
    - elvn[1,i],elvn[2,i],elvn[3,i],elvn[4,i],elvn[5,i],
      elvn[6,i], elvn[7,i],elvn[8,i],S; i = 1 to ne,
      enter 'F' for solid
  enter joint constraint(s) (in struc) - jnum,jdir, end = 0,0
      note: jdir may be 1,2,3 corresponding to the
            three coordinate-dir jdir may also be 0 for
            a fluid boundary
  enter joint coordinates (in prop) - x[1,j],x[2,j],x[3,j];
      j = 1 to nj
  enter element properties (in prop); i = 1 to ne:
      elastic modulus - emod[i]
      Poisson's ratio - nu[i]
      Density - ro[i]
      *** enter on single line as: emod[i],nu[i],ro[i]
  enter speed of sound in fluid (in prop); as a single value
      - speed
  enter structural damping value (in prop); as a single value
      - damp
  enter loading frequency (in main); as cycles/sec - freq
```

```

    enter joint reference load(s) (in load)
        - jnum,jdir,force, end = 0,0,0
    enter final time (in main) - tmax
    enter time step size (in main) - deltat
*/

// Define primary variables - number of elements, joints,
//and equations
long int ne, nef, nn, nj, njf, nie, nbc;
//nie- number of interface surfaces
long int neq, neqs, neqf;

// Define file pointers.
FILE *ifp; // Input file
FILE *ofp; // Output file

/* Define function prototypes - convention: "p" precedes the
name of the variable in main being pointed to in the function */

// Model definition functions:
int struc (long int *pjcode, long int *pjcodef, long int *pelvn,
           long int *pelnums, long int *pelnumf, long int *pinter,
           long int *pinterelms, long int *pinterelf,
           long int *pbnodes);
void codes (long int *pmcode, long int *pmcodef, long int *pjcode,
            long int *pjcodef, long int *pelvn, long int *pelnums,
            long int *pelnumf, long int *pinter);
void prop (double *px, double *pemod, double *pnu,
           double *pro, double *pc1, double *pc2, double *pc3,
           long int *pelvn, long int *pinterelms, long int
           *pinterelf, long int *pinter, double *pL, double *pA,
           double *pT, long int *pmcode,
           long int *pmcodef, long int *pjcode, long int *pjcodef,
           long int *pelnums, long int *pelnumf, long int *ptrans);
int load (double *pq, long int *pjcode);
void multip (double *pmatrix, double *parray, double *presult,
            long int eqns);
int solve(double *py, double *pA, double *px, int *pipiv);
void stiff (double *pssx, double *pemod, double *pnu, double *px,
            long int *pelvn, long int *pmcode, long int *plss,
            long int nelms, long int eqns, long int *pelnumx);
void stiffe (double *pK, double E, double nu, double *px,
            long int *pelvn, int n);
void mass1 (double *psmx, double *pro, double *px, long int *pelvn,

```

```

        long int *pelnumx, long int *pmcode, long int *plss,
        long int nelms, long int eqns);
void mass2 (double *psm, double *pro, double *px, long int *pelvn,
        long int *pmcode, double *pc1, double *pc2, double *pc3,
        long int nelms, long int eqns, long int *plss);
void mass3 (double *psmx, double *pro, double *px, long int *pelvn,
        long int *pelnumx, long int *pmcode, long int *plss,
        long int nelms, long int eqns);
void masse (double *pM, double ro, double *px, long int *pelvn,
        int n);
void damping (double *psd, double damp, double *psm1, double speed,
        long int *pelvn, long int *pbnodes, long int *pelnumf,
        double *pA, double *pbA, long int *pjcodef);
void transform (double *pkt, double *pT, double *pK, int n);
void jacobian (double *px, long int *pelvn, int n, int intpt,
        double *pJ, double *pJ_inv, double *pdetJ);

// Miscellaneous functions:
double shape (int n, int x, int intpt);
double volume (long int *pelvn, double *px, int n);
double dot (double *pa, double *pb, int n);
void cross (double *pa, double *pb, double *pc, int flag);
int closeio (int flag);

// Memory management functions:
double * alloc_dbl (long int size);
long int * alloc_int (long int size);
int * alloc_int2 (int size);
int free_all(double **pp2p2d, int nd, long int **pp2p2i, int ni,
        int **pp2p2i2, int ni2, int flag);

int main (void)
{
    // Open I/O for business!
    ifp = fopen(INPUT, "r"); // Open input file for reading
    ofp = fopen(OUTPUT, "w"); // Open output file for writing

    // Verify that I/O files are open
    if (ifp != 0) {
        printf("Input file is open\n");
    } else {
        printf("***ERROR*** Unable to open the input file\n");
        getchar();
        return 0;
    }
}

```

```

}
if (ofp != 0) {
    printf("Output file is open\n");
} else {
    printf("***ERROR*** Unable to open the output file\n");
    getchar();
    return 0;
}

// Read in number of elements, joints from input file
fscanf(ifs, "%ld\t%ld\t%ld\t%ld\t%ld\n", &ne, &nfe, &nn,
        &nie, &nbc);
fprintf(ofp, "Control Variables:\n\tNumber of Solid Elements:
        %ld\n", ne);
fprintf(ofp, "\n\tNumber of Fluid Elements: %ld\n", nfe);
fprintf(ofp, "\n\tNumber of Nodes Total: %ld\n", nn);
fprintf(ofp, "\n\tNumber of Interface Elements: %ld\n\n", nie);

// Memory management variables
double *p2p2d[ ];
int nd = 0;
long int *p2p2i[ ];
int ni = 0;
int *p2p2i2[ ];
int ni2 = 0;

// //Define secondary variables which DO NOT depend upon neq

// Current joint coordinates
double *x = alloc_dbl (3 * nn);
if (x == NULL) {
    return closeio(1);
}
p2p2d[nd] = x;
nd++;

// Element material properties
// Young's Moduli
double *emod = alloc_dbl (ne);
if (emod == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = emod;
nd++;

```

```

double *emodf = alloc_dbl (nef);
if (emod == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = emodf;
nd++;
// Poisson Ratios
double *nu = alloc_dbl (ne);
if (nu == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = nu;
nd++;
double *nuf = alloc_dbl (nef);
if (nuf == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = nuf;
nd++;
// Densities
double *ro = alloc_dbl (ne);
if (ro == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = ro;
nd++;
double *rof = alloc_dbl (nef);
if (rof == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = rof;
nd++;

double damp; // damping parameter
double speed; // speed of sound in fluid
double density; // density of fluid

// Direction cosines
double *c1 = alloc_dbl (3 * ne);
if (c1 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = c1;
nd++;

```

```

double *c2 = alloc_dbl (3 * ne);
if (c2 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = c2;
nd++;
double *c3 = alloc_dbl (3 * ne);
if (c3 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = c3;
nd++;

// Interface data vectors
// Interface nodes array
long int *inter = alloc_int (nie * 4);
if (inter == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = inter;
ni++;
// Solid interface solid elements array
long int *interelms = alloc_int (nie);
if (interelms == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = interelms;
ni++;
// Fluid interface elements array
long int *interelf = alloc_int (nie);
if (interelf == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = interelf;
ni++;
// Fluid boundary elements array
long int *bnodes = alloc_int (nbc);
if (bnodes == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = bnodes;
ni++;

// Element data vectors

```



```

// Member incidence data
long int *mcode = alloc_int (24 * ne);
if (mcode == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = mcode;
ni++;
long int *mcodef = alloc_int (24 * nef);
if (mcodef == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = mcodef;
ni++;
long int *jcode = alloc_int (3 * nn);
if (jcode == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = jcode;
ni++;
long int *jcodef = alloc_int (3 * nn);
if (jcodef == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = jcodef;
ni++;
long int *elvn = alloc_int (8 * (ne + nef));
if (elvn == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = elvn;
ni++;
long int *elnums = alloc_int (ne);
if (elnums == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = elnums;
ni++;
long int *elnumf = alloc_int (nef);
if (elnumf == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = elnumf;
ni++;

```

```

double deltat, tmax; // Max time and time step
int errchk; // Error check variable on user defined functions
double sum, A0, A1, A2, A3;
long int i, j, k; // Counter variables

// Pass control to model definition function
errchk = struc(jcode, jcodef, elvn, elnums, elnumf, inter,
               interelms, interelf, bnodes);

// Terminate program if errors encountered
if (errchk == 1) {
    fprintf(ofp, "\n\nSolution failed\n");
    printf("Solution failed, see output file\n");
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}

// Pass control to codes function
codes(mcode, mcodef, jcode, jcodef, elvn, elnums, elnumf,
      inter);

nj = neqs / 3.0 ;
njf = neqf;
fprintf(ofp, "\tNumber of Active Solid Nodes: %ld\n\n", nj);
fprintf(ofp, "\tNumber of Active Fluid Nodes: %ld\n\n", njf);
fprintf(ofp, "\tNumber of Interface Surfaces: %ld\n\n", nie);

// // Define secondary variables which D0 depend upon neq
// Generalized joint reference load vector
double *q = alloc_dbl (neq);
if (q == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = q;
nd++;
// Applied force vector
double *f = alloc_dbl (neq);
if (f == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = f;
nd++;
// Total and incremental generalized nodal displacement vectors
double *d = alloc_dbl (neq);
if (d == NULL) {

```

```

        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = d;
    nd++;
    // Total generalized nodal displacement vector from next time
    // step
    double *dd = alloc_dbl (neq);
    if (dd == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = dd;
    nd++;
    // Total generalized nodal displacement vector from previous
    //time step
    double *dp = alloc_dbl (neq);
    if (dp == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = dp;
    nd++;
    // Residual force vector, i.e. qtot - f
    double *r = alloc_dbl (neq);
    if (r == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = r;
    nd++;
    // Residual force vector at previous time
    double *rdp = alloc_dbl (neq);
    if (rdp == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = rdp;
    nd++;
    // Residual force vector at next time step
    double *rd = alloc_dbl (neq);
    if (rd == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = rd;
    nd++;
    // Effective residual force vector
    double *reff = alloc_dbl (neq);
    if (reff == NULL) {

```

```

        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = reff;
    nd++;
    // Effective mass vector
    double *Meff = alloc_dbl (neq*neq);
    if (Meff == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = Meff;
    nd++;
    // Accelerations at time t
    double *accel = alloc_dbl (neq);
    if (accel == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = accel;
    nd++;
    // Velocities at time t
    double *vel = alloc_dbl (neq);
    if (vel == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = vel;
    nd++;

    /* Note: the mass matrix and damping arrays may be unlumped and
    longer than neq if the damping matrix must be non-diagonal */
    // Combined mass matrix array
    double *sm = alloc_dbl (neq*neq);
    if (sm == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = sm;
    nd++;
    // Solid mass array
    double *sm1 = alloc_dbl (neqs*neqs);
    if (sm1 == NULL) {
        return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
    }
    p2p2d[nd] = sm1;
    nd++;
    // Fluid mass array
    double *sm2 = alloc_dbl (neqf*neqf);

```

```

if (sm2 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = sm2;
nd++;
// General damping array
double *sd = alloc_dbl (neq*neq);
if (sd == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = sd;
nd++;

long int lss;
long int lssf;
lss = lssf = 0;
lss = neqs * neqs;

// Pass control to skylin function for fluid
//skylin (khtf, maxaf, mcodef, &lssf, 1);
lssf = neqf * neqf;

printf("\nNum of Eqns: %ld\n", neq);
// Define secondary variables which depend upon lss
// General stiffness array
double *ss = alloc_dbl (neq * neq);
if (ss == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = ss;
nd++;
// Solid stiffness array
double *ss1 = alloc_dbl (lss);
if (ss1 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = ss1;
nd++;
// Fluid stiffness array
double *ss2 = alloc_dbl (lssf);
if (ss2 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = ss2;

```

```

nd++;
// Interface transformation-area matrix
double *L = alloc_dbl (neqs * neqf);
if (L == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = L;
nd++;
// Interface area matrix
double *A = alloc_dbl (neqf * neqf);
if (A == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = A;
nd++;
// Interface transform matrix
double *T = alloc_dbl (neqs * neqf);
if (T == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = T;
nd++;
// Internal force vector 1
double *r1 = alloc_dbl (neq);
if (r1 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = r1;
nd++;
// Internal force vector 2
double *r2 = alloc_dbl (neq);
if (r2 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = r2;
nd++;
// Internal force vector 3
double *r3 = alloc_dbl (neq);
if (r3 == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = r3;
nd++;
// Boundary area matrix

```

```

double *bA = alloc_dbl (neqf*neqf);
if (bA == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2d[nd] = bA;
nd++;
// Storage vector for Lapack solver
int *ipiv = alloc_int2 (neq);
if (ipiv == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i2[ni1] = ipiv;
ni2++;
// Element transformation surface data vector
long int *trans = alloc_int (nie);
if (trans == NULL) {
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}
p2p2i[ni] = trans;
ni++;

// Pass control to prop function
prop (x, emod, nu, ro, c1, c2, c3, elvn, interelms,
      interelf, inter, L, A, T, mcode, mcodef, jcode, jcodef,
      elnums, elnumf, trans);

// Scan speed of sound in fluid
fscanf(ifp, "%lf\n", &speed);
printf("\nSpeed of Sound in Fluid: %lf\n", speed);

// Scan density of fluid
fscanf(ifp, "%lf\n", &density);
printf("\nDensity of Fluid: %lf\n", density);

// Scan structural damping value
fscanf(ifp, "%lf\n", &damp);
printf("\nDamping value: %lf\n", damp);

double freq;
// Scan loading frequency
fscanf(ifp, "%lf\n", &freq);
printf("\nLoading Frequency: %lf\n", freq);

// Pass control to load function

```

```

errchk = load (q, jcode);
// Terminate program if errors encountered
if (errchk == 1)
{
    fprintf(ofp, "\n\nSolution failed\n");
    printf("Solution failed, see output file\n");

    goto term_main;
}

// Read in solver parameters from input file
fscanf(ifp, "%le\t%le\n", &tmax, &deltat);
fprintf(ofp, "\ntmax is %le", tmax);
fprintf(ofp, "\ndeltat is %le", deltat);

// Print layout for output of results
fprintf(ofp, "\nTime Stepping:\n\tTime #\t\tLoad\t");

// 1. Form Stiffness, Mass, and Damping Matrices //
// Initialize effective mass array to zero
for (i = 0; i <= neq-1; ++i) {
    for (j = 0; j <= neq-1; ++j) {
        Meff[i*neq + j] = 0;
    }
}

// Define fluid properties
if(nef != 0){
for(i = 0; i <= nef - 1; ++i){
    rof[i] = 1/(speed*speed);
    emodf[i] = 10E20;
    nuf[i] = 10E20/2;
}
}

// Pass control to stiff function
stiff (ss1, emod, nu, x, elvn, mcode, &lss, ne, neqs,
        elnums);

//Assemble combined stiffness matrix
int a;
int b;
for(i = 0; i <= neq-1; ++i){
    for(j = 0; j <= neq-1; ++j){

```



```

        a = i;
        b = j;
        ss[a*neq + b] = 0;
    }
}
for(i = 0; i <= neqs-1; ++i){
    for(j = 0; j <= neqs-1; ++j){
        a = i;
        b = j;
        ss[a*neq + b] = ss1[i*neqs + j];
    }
}
for(i = 0; i <= neqs-1; ++i){
    for(j = 0; j <= neqf-1; ++j){
        a = i;
        b = j + neqs;
        ss[(a*neq) + b] = L[i*neqf + j]*1;
    }
}

stiff (ss2, emodf, nuf, x, elvn, mcodef, &lssf, nef,
      neqf, elnumf);

for(i = 0; i <= neqf-1; ++i){
    for(j = 0; j <= neqf-1; ++j){
        a = i + neqs;
        b = j + neqs;
        ss[(a*neq) + b] = ss2[i*neqf + j];
    }
}

// Pass control to mass function
// 1. Use the row-lumped consistent mass matrix.
// Include fixed dofs.
// mass1 (sm1, ro, x, elvn, elnums, mcode, &lss, ne, neqs);
// mass1 (sm2, rof, x, elvn, elnumf, mcodef, &lssf, nef,
//      neqf);
// 2. Use the density-volume diagonal mass matrix.
// mass2 (sm1, ro, x, elvn, mcode, c1, c2, c3, ne, neqs,
//      &lss);
// mass2 (sm2, rof, x, elvn, mcodef, c1, c2, c3, nef, neqf,
//      &lssf);
// 3. Use the row-lumped consistent mass matrix.
// Exclude fixed dofs.

```

```

    // mass3 (sm1, ro, x, elvn, elnums, mcode, &lss, ne, neqs);
    // mass3 (sm2, rof, x, elvn, elnumf, mcodef, &lssf, nef,
    // neqf);

    mass1 (sm1, ro, x, elvn, elnums, mcode, &lss, ne, neqs);

//    //Assemble combined mass matrix
    for(i = 0; i <= neq-1; ++i){
        for(j = 0; j <= neq-1; ++j){
            sm[i*neq + j] = 0;
        }
    }
    for(i = 0; i <= neqs-1; ++i){
        for(j = 0; j <= neqs-1; ++j){
            sm[i*neq + j] = sm1[i*neqs + j];
        }
    }
    for(i = 0; i <= neqf-1; ++i){
        for(j = 0; j <= neqs-1; ++j){
            sm[((i+neqs)*neq) + j] =
                (-1.0 * density * L[(j*neqf) + i]);
        }
    }

    mass1 (sm2, rof, x, elvn, elnumf, mcodef, &lssf, nef, neqf);

    for(i = 0; i <= neqf-1; ++i){
        for(j = 0; j <= neqf-1; ++j){
            sm[((i+neqs)*neq) + (j+neqs)] = sm2[i*neqf + j];
        }
    }

    // Pass control to damping function
    //damping (sd, damp, sm1, speed, elvn, bnodes, elnumf,
    //          A, bA, jcodef);

// 2. Initialize time iteration
// Initialize d, v, a at time zero
    for (i = 0; i <= neq - 1; ++i) {
        d[i] = vel[i] = accel[i] = 0;
    }
// Calculate integration constants
    A0 = 1.0 / (deltat * deltat);
    A1 = 1.0 / (2.0 * deltat);

```

```

A2 = 2.0 * A0;
A3 = 1.0 / A2;

// Calculate previous displacement at time -deltat
for (i = 0; i <= neq - 1; ++i) {
    dp[i] = d[i] - (deltat * vel[i]) + (A3 * accel[i]);
}

// Form the effective mass matrix Meff = a0*M + a1*C
for (i = 0; i <= neq - 1; ++i){
    for (j = 0; j <= neq - 1; ++j){
        Meff[i*neq + j] = (A0 * sm[i*neq + j]);
    }
}

// 3. Loop over all time steps until final time step is reached
// fprintf(ofp, "tmax: %ld\n", tmax);
double itecnt;
itecnt = 0;
do {

    // Calculate the effective loads at time t
    sum = 1;
    for( i = 0; i <= neq - 1; ++i){
        f[i] = 0;
        r1[i] = 0;
        r2[i] = 0;
    }

    // Calculate internal force vector at time t
    multip (ss, d, f, neq);
    multip (sm, d, r1, neq);
    multip (sm, dp, r2, neq);

    // Specify loading function:
    // 1. For ramped loading:
    //if (itecnt < 500){ sum = itecnt/500; }
    // 2. For constant loading :
    if (itecnt >= 3420){ sum = 0; }
    // 3. For sinusoidal loading:
    if (itecnt < 3420) { sum = sin((2 *
        3.1415926535897932384626433832795) *
        freq * (itecnt * deltat))*

```

```

        sin((2 * 3.1415926535897932384626433832795) *
            (25) * (itecnt * deltat)); }

// Compute interface forces due to fluid
// For solid nodes: -1*L*p_incident
// (ie. L * (d at interface nodes))
// This is zero for now since we have no incident pressures
// For fluid nodes: -1*ro*A*a_normal_incident
// This is also zero for now since we have no incident
// acceleration

for (i = 0; i <= neq - 1; ++i){
    reff[i] = (q[i]*sum) - f[i] + (A2 * r1[i]) -
        (A0 * r2[i]);
}

fprintf(ofp, "%1E ", (itecnt*deltat));
fprintf(ofp, " %1E ", (-0.444444444444*sum));

// Solve for the displacements at time t+deltat
// Meff*dd = reff (A*x = y)
printf("timestep %lf / %lf : ", itecnt, (tmax/deltat));
errchk = solve(reff, Meff, dd, ipiv);
// Terminate program if errors encountered
if (errchk == 1) {
    fprintf(ofp, "\n\nSolution failed\n");
    printf("Solution failed, see output file\n");
    // free_dbl all allocated memory
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}

}

fprintf(ofp, "\n");

// Caluculate accelerations and velocities at time t
for (i = 0; i <= neq - 1; ++i){
    accel[i] = A0 * (dp[i] - (2.0 * d[i]) + dd[i]);
    vel[i] = A1 * ((-1 * dp[i]) + dd[i]);
}

// Output model response
fprintf(ofp, "\ndisp: %1G\t%lf\t%lf ", itecnt,
        itecnt*deltat, (q[0] * sum));
for (i = 0; i <= neq - 1; ++i) {

```

```

        fprintf(ofp, "%lE, ", d[i]);
    }

    fprintf(ofp, "\n\nvel:          %lf\t%d\t", itecnt*deltat,
              itecnt);
    for (i = 0; i <= neq - 1; ++i) {
        fprintf(ofp, "\t%lf", vel[i]);
    }
    fprintf(ofp, "\n\naccel:      %lf\t%d\t", itecnt*deltat,
              itecnt);
    for (i = 0; i <= neq - 1; ++i) {
        fprintf(ofp, "\t%lf", accel[i]);
    }

    // Increment values to next time step
    for (i = 0; i <= neq - 1; ++i){
        dp[i] = d[i];
        d[i] = dd[i];
    }
    itecnt = itecnt + 1;

} while ((itecnt * deltat) <= tmax);

fprintf(ofp, "\n\nSolution successful\n");

printf("Solution successful\n");

term_main:
    // Free all allocated memory
    printf("Exiting Main Function\n");
    return free_all(p2p2d, nd, p2p2i, ni, p2p2i2, ni2, 1);
}

/* This function reads in member incidences and joint constraints
and checks for errors in the joint constraint input */
int struc (long int *pjcode, long int *pjcodef, long int *pelvn,
           long int *pelnums, long int *pelnumf, long int *pinter,
           long int *pinterelms, long int *pinterelmf,
           long int *pbnodes)
{
    long int i, j, k, l, m, n, o, p, q;
    long int r, s, t;
    int count1, count2, count3, count4;
    int c1, c2, c3, c4;

```



```

        *(pelvn+i*8+2), *(pelvn+i*8+3), *(pelvn+i*8+4),
        *(pelvn+i*8+5), *(pelvn+i*8+6), *(pelvn+i*8+7),
        type);
    }

// Define interface nodes and elements
if (nef != 0){
    c1 = c2 = c3 = c4 = 0;
    for (i = 0; i <= ne - 1; ++i) {
        elm_num = *(pelnums + i) - 1;
        j = *(pelvn + elm_num*8);
        k = *(pelvn + elm_num*8 + 1);
        l = *(pelvn + elm_num*8 + 2);
        m = *(pelvn + elm_num*8 + 3);
        n = *(pelvn + elm_num*8 + 4);
        o = *(pelvn + elm_num*8 + 5);
        p = *(pelvn + elm_num*8 + 6);
        q = *(pelvn + elm_num*8 + 7);
        for (r = 0; r <= nef - 1; ++r) {
            c1 = 0;
            c3 = 0;
            elm_numf = *(pelnumf + r) - 1 ;
            for (s = 0; s <= 7; ++s){
                if(j == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = j;
                ++c3;}
                if(k == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = k;
                ++c3;}
                if(l == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = l;
                ++c3;}
                if(m == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = m;
                ++c3;}
                if(n == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = n;
                ++c3;}
                if(o == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = o;
                ++c3;}
                if(p == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = p;
                ++c3;}
                if(q == *(pelvn+elm_numf*8+s)){++c1; surf[c3] = q;
                ++c3;}
            }
            if(c1 == 4){
                *(pinterelf + c4) = elm_numf + 1;
                *(pinterelms + c4) = elm_num + 1;
                for(t = 0; t <= 3; ++t){

```

```

                *(pinter + c4*4 + t) = surf[t];
            }
            ++c4;
        }
    }
} //Close If statement

// Read in fluid boundary nodes
fscanf(ifp, "%ld\n", &j);
fprintf(ofp, "\nBoundary Nodes:\n");
i = 0;
while ((j != 0))
{
    *(pbnodes + i) = j;
    fprintf(ofp, "\t%ld\n", j);
    ++i;
    fscanf(ifp, "%ld\n", &j);
}

// Establish joint constraints
fprintf(ofp, "\nJoint Constraints:\n\tJoint\tDirection\n");
for (i = 0; i <= (3 * nn) - 1; ++i) {
    *(pjcode+i) = 1; // Initialize jcode with ones
    if(ne == 0){*(pjcode+i) = 0;}
}

// Establish joint constraints
for (i = 0; i <= (3 * nn) - 1; ++i) {
    *(pjcode+i) = 1; // Initialize jcode with ones
    if(nef == 0){*(pjcode+i) = 0;}
}

// Read in joint number and constraint direction from input file
fscanf(ifp, "%ld\t%ld\n", &j, &k);
while ((j != 0) && (k != 0))
{
    switch (k) {
        case 1:
            *(pjcode+(j-1)*3+(k-1)) = 0;
            fprintf(ofp, "\t%ld\t%ld\n", j, k);
            break;
        case 2:
            *(pjcode+(j-1)*3+(k-1)) = 0;
            fprintf(ofp, "\t%ld\t%ld\n", j, k);

```



```

        break;
    case 3:
        *(pjcode+(j-1)*3+(k-1)) = 0;
        fprintf(ofp, "\t%ld\t%ld\n", j, k);
        break;
    default:
        fprintf(ofp, "\n***ERROR*** Joint constraint"
            "input not recognized");
        return 1;
        break;
    }
    fscanf(Ifp, "%ld\t%ld\n", &j, &k);
}

// Zero out fluid boundary dofs
for (i = 0; i <= nbc - 1 ; ++i) {
    j = *(pbnodes + i);
    *(pjcodef+(j-1)*3) = 0;
    fprintf(ofp, "\t%ld\tzero p\n", j);
}

printf("Exiting Struc Function\n");
return 0;
}

/* This function generates joint code, jcode, by assigning integers
in sequence, by columns, to all nonzero elements of jcode from 1 to
neq; generate the member code, mcode, by transferring, via elvn,
columns of jcode into columns of mcode. Follows procedure in
Chapter 6 of Bathe and Wilson as well as Chapter 6 of Holzer */
void codes (long int *pmcode, long int *pmcodef, long int *pjcode,
            long int *pjcodef, long int *pelvn, long int *pelnums,
            long int *pelnumf, long int *pinter)
{
    long int i, j, k, l, m, n, o, p, q, r, t;
    // Initialize function variables
    int elm_num, zero;
    int zeroj, zerok, zerol, zerom, zeron, zeroo, zerop, zeroq;

    // Zero out all nodes in jcode that are not attached
    //to a solid element
    // aka. are part of a fluid element and NOT on the interface.
    if (nef != 0) {
        for (i = 0; i <= nef - 1; ++i) {

```

```

    elm_num = *(pelnumf + i) - 1;
    j = *(pelvn+elm_num*8); // Establish Vertex 1 joint number
    zeroj = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(j == *(pinter + r))
    {zeroj = 0;}}
    k = *(pelvn+elm_num*8+1); // Establish Vertex 2 joint number
    zerok = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(k == *(pinter + r))
    {zerok = 0;}}
    l = *(pelvn+elm_num*8+2); // Establish Vertex 3 joint number
    zerol = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(l == *(pinter + r))
    {zerol = 0;}}
    m = *(pelvn+elm_num*8+3); // Establish Vertex 4 joint number
    zerom = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(m == *(pinter + r))
    {zerom = 0;}}
    n = *(pelvn+elm_num*8+4); // Establish Vertex 5 joint number
    zeron = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(n == *(pinter + r))
    {zeron = 0;}}
    o = *(pelvn+elm_num*8+5); // Establish Vertex 6 joint number
    zeroo = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(o == *(pinter + r))
    {zeroo = 0;}}
    p = *(pelvn+elm_num*8+6); // Establish Vertex 7 joint number
    zerop = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(p == *(pinter + r))
    {zerop = 0;}}
    q = *(pelvn+elm_num*8+7); // Establish Vertex 8 joint number
    zeroq = 1;
    for(r = 0; r <= (4*nle) - 1; ++r){if(q == *(pinter + r))
    {zeroq = 0;}}
    for (t = 0; t <= 2; ++t) {
        if(zeroj == 1){*(pjcode+(j-1)*3+t) = 0;}
        if(zerok == 1){*(pjcode+(k-1)*3+t) = 0;}
        if(zerol == 1){*(pjcode+(l-1)*3+t) = 0;}
        if(zerom == 1){*(pjcode+(m-1)*3+t) = 0;}
        if(zeron == 1){*(pjcode+(n-1)*3+t) = 0;}
        if(zeroo == 1){*(pjcode+(o-1)*3+t) = 0;}
        if(zerop == 1){*(pjcode+(p-1)*3+t) = 0;}
        if(zeroq == 1){*(pjcode+(q-1)*3+t) = 0;}
    }
}

```

```

// Zero out all nodes in jcodef that are not attached to a fluid
// element
// aka. are part of a solid element and NOT on the interface.
for (i = 0; i <= ne - 1; ++i) {
    elm_num = *(pelnums + i) - 1;
    j = *(pelvn+elm_num*8); // Establish Vertex 1 joint number
    zeroj = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(j == *(pinter + r)){zeroj = 0;}}
    k = *(pelvn+elm_num*8+1); // Establish Vertex 2 joint number
    zerok = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(k == *(pinter + r)){zerok = 0;}}
    l = *(pelvn+elm_num*8+2); // Establish Vertex 3 joint number
    zerol = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(l == *(pinter + r)){zerol = 0;}}
    m = *(pelvn+elm_num*8+3); // Establish Vertex 4 joint number
    zerom = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(m == *(pinter + r)){zerom = 0;}}
    n = *(pelvn+elm_num*8+4); // Establish Vertex 5 joint number
    zeron = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(n == *(pinter + r)){zeron = 0;}}
    o = *(pelvn+elm_num*8+5); // Establish Vertex 6 joint number
    zeroo = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(o == *(pinter + r)){zeroo = 0;}}
    p = *(pelvn+elm_num*8+6); // Establish Vertex 7 joint number
    zerop = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(p == *(pinter + r)){zerop = 0;}}
    q = *(pelvn+elm_num*8+7); // Establish Vertex 8 joint number
    zeroq = 1;
    for(r = 0; r <= (4*nie) - 1; ++r)
        {if(q == *(pinter + r)){zeroq = 0;}}
    for (t = 0; t <= 2; ++t) {
        if(zeroj == 1){*(pjcodef+(j-1)*3+t) = 0;}
        if(zerok == 1){*(pjcodef+(k-1)*3+t) = 0;}
        if(zerol == 1){*(pjcodef+(l-1)*3+t) = 0;}
        if(zerom == 1){*(pjcodef+(m-1)*3+t) = 0;}
        if(zeron == 1){*(pjcodef+(n-1)*3+t) = 0;}
    }
}

```

```

        if(zeroo == 1){*(pjcodef+(o-1)*3+t) = 0;}
        if(zerop == 1){*(pjcodef+(p-1)*3+t) = 0;}
        if(zeroq == 1){*(pjcodef+(q-1)*3+t) = 0;}
    }
}
} // End If statement

neq = neqs = neqf = 0;
// Assign solid equation numbers first
for (i = 0; i <= nn - 1; ++i) {
    for(j = 0; j <= 2; ++j) {
        if(*(pjcode+i*3+j) != 0) {
            ++neq;
            *(pjcode+i*3+j) = neq;
            ++neqs;
        }
    }
}
// Assign fluid equation numbers second
for (i = 0; i <= nn - 1; ++i) {
    if(*(pjcodef+i*3) != 0) {
        ++neq;
        *(pjcodef+i*3) = neq;
        *(pjcodef+i*3+1) = 0;
        *(pjcodef+i*3+2) = 0;
        ++neqf;
    }
}

// Generate mcode from jcode using member incidence matrix
for (i = 0; i <= ne - 1; ++i) {
    elm_num = *(pelnums + i) - 1;
    // Establish joint numbers
    j = *(pelvn+elm_num*8);
    k = *(pelvn+elm_num*8+1);
    l = *(pelvn+elm_num*8+2);
    m = *(pelvn+elm_num*8+3);
    n = *(pelvn+elm_num*8+4);
    o = *(pelvn+elm_num*8+5);
    p = *(pelvn+elm_num*8+6);
    q = *(pelvn+elm_num*8+7);
    // Increment of DOFs for each vertex
    for (t = 0; t <= 2; ++t) {
        *(pmcode+(i*24)+t) = *(pjcode+(j-1)*3+t);
    }
}

```

```

        *(pmcode+(i*24)+3+t) = *(pjcode+(k-1)*3+t);
        *(pmcode+(i*24)+6+t) = *(pjcode+(l-1)*3+t);
        *(pmcode+(i*24)+9+t) = *(pjcode+(m-1)*3+t);
        *(pmcode+(i*24)+12+t) = *(pjcode+(n-1)*3+t);
        *(pmcode+(i*24)+15+t) = *(pjcode+(o-1)*3+t);
        *(pmcode+(i*24)+18+t) = *(pjcode+(p-1)*3+t);
        *(pmcode+(i*24)+21+t) = *(pjcode+(q-1)*3+t);
    }
}
if (nef != 0){
for (i = 0; i <= nef - 1; ++i) {
    elm_num = *(pelnumf + i) - 1;
    j = *(pelvn+elm_num*8);
    k = *(pelvn+elm_num*8+1);
    l = *(pelvn+elm_num*8+2);
    m = *(pelvn+elm_num*8+3);
    n = *(pelvn+elm_num*8+4);
    o = *(pelvn+elm_num*8+5);
    p = *(pelvn+elm_num*8+6);
    q = *(pelvn+elm_num*8+7);
    // Increment of DOFs for each vertex
    for (t = 0; t <= 2; ++t) {
        *(pmcodef+(i*24)+t) = *(pjcodef+(j-1)*3+t);
        *(pmcodef+(i*24)+3+t) = *(pjcodef+(k-1)*3+t);
        *(pmcodef+(i*24)+6+t) = *(pjcodef+(l-1)*3+t);
        *(pmcodef+(i*24)+9+t) = *(pjcodef+(m-1)*3+t);
        *(pmcodef+(i*24)+12+t) = *(pjcodef+(n-1)*3+t);
        *(pmcodef+(i*24)+15+t) = *(pjcodef+(o-1)*3+t);
        *(pmcodef+(i*24)+18+t) = *(pjcodef+(p-1)*3+t);
        *(pmcodef+(i*24)+21+t) = *(pjcodef+(q-1)*3+t);
    }
}
} //Close If statement

fprintf(ofp, "\nNumber of equations (system DOFs): %ld\n\n",
        neq);
printf("Exiting Codes Function\n");
}

/* This function reads and echoes joint coordinates, element
properties, as well as computing element side-lengths, areas, and
direction cosines */
void prop (double *px, double *pemod, double *pnu, double *pro,
double *pc1, double *pc2, double *pc3, long int *pelvn,

```

```

    long int *pinterelms, long int *pinterelf, long int *pinter,
    double *pL, double *pA, double *pT, long int *pmcode,
    long int *pmcodef, long int *pjcode, long int *pjcodef,
    long int *pelnums, long int *pelnumf, long int *ptrans)
{
    // Initialize function variables
    long int i, r, s, t, u, a, b;
    int node1, node2, node3, node4;
    double sum, interarea;
    int elm_num, elm_numf, elm_value, elm_valuef;
    double x1, x2, x3;
    double el12[3], el23[3], el34[3], el41[3], el26[3], el37[3],
        el48[3], el15[3], el56[3], el67[3], el78[3], el85[3],
        eln12[3], eln13[3], normal[3];
    double lengthn12, lengthn13;
    double xemod, xnu, xarea, xro;
    double localx[3], localy[3], localz[3];
    int u1, u2, u3, u4, s1, s2, s3, s4, value[4];
    int coord1, coord2, coord3, coord4, nodeA, nodeB, nodeC;

    fprintf(ofp, "Joint Coordinates:\n\tJoint\tDirection-1\t"
        "Direction-2\tDirection-3\n");

    for(r = 0; r <= neqf - 1; ++r){
        for(t = 0; t <= neqf - 1; ++t){
            *(pA + r*neqf + t) = 0;
        }
    }
    for(r = 0; r <= neqs - 1; ++r){
        for(t = 0; t <= neqf - 1; ++t){
            *(pT + r*neqf + t) = 0;
        }
    }

    for (i = 0; i <= nn - 1; ++i)
    {
        // Read in joint coordinates from input file
        fscanf(ifp, "%lf\t%lf\t%lf\n", &x1, &x2, &x3);
        *(px+i*3) = x1;
        *(px+i*3+1) = x2;
        *(px+i*3+2) = x3;

        fprintf(ofp, "\t%ld\t%lf\t%lf\t%lf\n", i + 1, *(px+i*3),
            *(px+i*3+1), *(px+i*3+2));
    }
}

```

```

}

fprintf(ofp, "\nElement Properties:\n\tElement\t\t"
        "Elastic Modulus\t\t");
fprintf(ofp, "Poisson's Ratio\t\tDensity\n");
//fprintf(ofp, "\nne: %d\n", ne);

fscanf(ifp, "%lf\t%lf\t%lf\n", &xemod, &xnu, &xro);

/* Read and echo solid element properties and compute element
   side-lengths, area, direction cosines, and non-zero initial
   element coordinates */
for (i = 0; i <= ne - 1; ++i)
{
    // Read and echo element properties
    //fscanf(ifp, "%lf\t%lf\t%lf\n", &xemod, &xnu, &xro);
    *(pemod+i) = xemod;
    *(pnu+i) = xnu;
    *(pro+i) = xro;

    fprintf(ofp, "\t%ld\t\t%lE\t\t%lf\t\t%lf\n", i + 1,
            *(pemod+i), *(pnu+i), *(pro+i));
}

/* Now we identify interface areas by running through the solid
   elements*/
if (nie != 0){
for (i = 0; i <= nie - 1; ++i)
{
    elm_num = *(pinterelms + i) - 1;
    elm_value = 0;
    for(r = 0; r <= ne - 1; ++r){
        if(*(pelnums + r) == elm_num + 1){elm_value = r;}
    }
    elm_numf = *(pinterelf + i) - 1;
    elm_valuef = 0;
    for(r = 0; r <= nef - 1; ++r){
        if(*(pelnumf + r) == elm_numf + 1){elm_valuef = r;}
    }

    // do the cross of the two vectors created by those four
    // nodes to get the normal vector

    // The nodes are assigned to the element orientation:

```

```

nodeA = nodeB = nodeC = 0;
node1 = *(pinter + i*4);
node2 = *(pinter + i*4 + 1);
node3 = *(pinter + i*4 + 2);
node4 = *(pinter + i*4 + 3);
for(r = 0; r <= 7; ++r){
    if(*(pelvn + elm_num*8 + r) == node1){coord1 = r+1;}
    if(*(pelvn + elm_num*8 + r) == node2){coord2 = r+1;}
    if(*(pelvn + elm_num*8 + r) == node3){coord3 = r+1;}
    if(*(pelvn + elm_num*8 + r) == node4){coord4 = r+1;}
}
value[0] = 1; value[1] = 2; value[2] = 3; value[3] = 4;
s = 0;
nodeA = 0; nodeB = 0; nodeC = 0;
for( r = 0; r <= 3; ++r){
    if(coord1 == value[r]){++s;}
    if(coord2 == value[r]){++s;}
    if(coord3 == value[r]){++s;}
    if(coord4 == value[r]){++s;}
}
if(s != 4){s = 0;}
if(s == 4){nodeA = node1; nodeB = node2; nodeC = node4,
    *(ptrans + i) = 1;}

value[0] = 1; value[1] = 2; value[2] = 5; value[3] = 6;
s = 0;
for( r = 0; r <= 3; ++r){
    if(coord1 == value[r]){++s;}
    if(coord2 == value[r]){++s;}
    if(coord3 == value[r]){++s;}
    if(coord4 == value[r]){++s;}
}
if(s != 4){s = 0;}
if(s == 4){nodeA = node3; nodeB = node4; nodeC = node1,
    *(ptrans + i) = 2;}

value[0] = 1; value[1] = 4; value[2] = 5; value[3] = 8;
s = 0;
for( r = 0; r <= 3; ++r){
    if(coord1 == value[r]){++s;}
    if(coord2 == value[r]){++s;}
    if(coord3 == value[r]){++s;}
    if(coord4 == value[r]){++s;}
}

```



```

if(s != 4){s = 0;}
if(s == 4){nodeA = node3; nodeB = node1; nodeC = node4,
    *(ptrans + i) = 3;}

value[0] = 2; value[1] = 3; value[2] = 6; value[3] = 7;
s = 0;
for( r = 0; r <= 3; ++r){
    if(coord1 == value[r]){++s;}
    if(coord2 == value[r]){++s;}
    if(coord3 == value[r]){++s;}
    if(coord4 == value[r]){++s;}
}
if(s != 4){s = 0;}
if(s == 4){nodeA = node1; nodeB = node3; nodeC = node2,
    *(ptrans + i) = 4;}

value[0] = 5; value[1] = 6; value[2] = 7; value[3] = 8;
s = 0;
for( r = 0; r <= 3; ++r){
    if(coord1 == value[r]){++s;}
    if(coord2 == value[r]){++s;}
    if(coord3 == value[r]){++s;}
    if(coord4 == value[r]){++s;}
}
if(s != 4){s = 0;}
if(s == 4){nodeA = node2; nodeB = node1; nodeC = node3,
    *(ptrans + i) = 5;}

value[0] = 3; value[1] = 4; value[2] = 7; value[3] = 8;
s = 0;
for( r = 0; r <= 3; ++r){
    if(coord1 == value[r]){++s;}
    if(coord2 == value[r]){++s;}
    if(coord3 == value[r]){++s;}
    if(coord4 == value[r]){++s;}
}
if(s != 4){s = 0;}
if(s == 4){nodeA = node2; nodeB = node1; nodeC = node4,
    *(ptrans + i) = 6;}

if(nodeA == 0 && nodeB == 0 && nodeC == 0)
    {fprintf(ofp, "\n, **Error** in interface surface"
        " determination\n");}

```

```

// Side AB
eln12[0] = *(px+(nodeA-1)*3) - *(px+(nodeB-1)*3);
eln12[1] = *(px+(nodeA-1)*3+1) - *(px+(nodeB-1)*3+1);
eln12[2] = *(px+(nodeA-1)*3+2) - *(px+(nodeB-1)*3+2);
lengthn12 = sqrt(dot (eln12, eln12, 3));
// Side AC
eln13[0] = *(px+(nodeA-1)*3) - *(px+(nodeC-1)*3);
eln13[1] = *(px+(nodeA-1)*3+1) - *(px+(nodeC-1)*3+1);
eln13[2] = *(px+(nodeA-1)*3+2) - *(px+(nodeC-1)*3+2);
lengthn13 = sqrt(dot (eln13, eln13, 3));
// Compute vector normal to interface surface
cross (eln12, eln13, normal, 0);

// compute the area
interarea = sqrt(dot (normal, normal, 3));

// compute the direction cosines
localx[0] = eln12[0] / lengthn12;
localx[1] = eln12[1] / lengthn12;
localx[2] = eln12[2] / lengthn12;
localz[0] = normal[0] / interarea;
localz[1] = normal[1] / interarea;
localz[2] = normal[2] / interarea;
cross (localz, localx, localy, 1);
for (t = 0; t <= 2; ++t)
{
    *(pc1+elm_value*3+t) = localx[t];
    *(pc2+elm_value*3+t) = localy[t];
    *(pc3+elm_value*3+t) = localz[t];
}

// assign the area to the correct location in the (neqfxneqf)
// interface area matrix
u1 = *(pjcodef + (node1-1)*3) - neqs;
u2 = *(pjcodef + (node2-1)*3) - neqs;
u3 = *(pjcodef + (node3-1)*3) - neqs;
u4 = *(pjcodef + (node4-1)*3) - neqs;
*(pA + (u1-1)*neqf + (u1-1)) += interarea;
*(pA + (u2-1)*neqf + (u2-1)) += interarea;
*(pA + (u3-1)*neqf + (u3-1)) += interarea;
*(pA + (u4-1)*neqf + (u4-1)) += interarea;

// assign the transformation for this surface
// zero out transformation matrix first

```

```

for(r = 0; r <= neqs - 1; ++r){
    for(s = 0; s <= neqf - 1; ++s){
        *(pT + r*neqf + s) = 0;
    }
}

if( *(ptrans + i) == 1 ){
    s1 = *(pjcode + (node1-1)*3 + 0); if(s1 != 0)
    { *(pT + (s1-1)*neqf + (u1-1)) = 1; }
    s2 = *(pjcode + (node2-1)*3 + 0); if(s2 != 0)
    { *(pT + (s2-1)*neqf + (u2-1)) = 1; }
    s3 = *(pjcode + (node3-1)*3 + 0); if(s3 != 0)
    { *(pT + (s3-1)*neqf + (u3-1)) = 1; }
    s4 = *(pjcode + (node4-1)*3 + 0); if(s4 != 0)
    { *(pT + (s4-1)*neqf + (u4-1)) = 1; }
}

if( *(ptrans + i) == 2 ){
    s1 = *(pjcode + (node1-1)*3 + 1); if(s1 != 0)
    { *(pT + (s1-1)*neqf + (u1-1)) = 1; }
    s2 = *(pjcode + (node2-1)*3 + 1); if(s2 != 0)
    { *(pT + (s2-1)*neqf + (u2-1)) = 1; }
    s3 = *(pjcode + (node3-1)*3 + 1); if(s3 != 0)
    { *(pT + (s3-1)*neqf + (u3-1)) = 1; }
    s4 = *(pjcode + (node4-1)*3 + 1); if(s4 != 0)
    { *(pT + (s4-1)*neqf + (u4-1)) = 1; }
}

if( *(ptrans + i) == 3 ){
    s1 = *(pjcode + (node1-1)*3 + 2); if(s1 != 0)
    { *(pT + (s1-1)*neqf + (u1-1)) = -1; }
    s2 = *(pjcode + (node2-1)*3 + 2); if(s2 != 0)
    { *(pT + (s2-1)*neqf + (u2-1)) = -1; }
    s3 = *(pjcode + (node3-1)*3 + 2); if(s3 != 0)
    { *(pT + (s3-1)*neqf + (u3-1)) = -1; }
    s4 = *(pjcode + (node4-1)*3 + 2); if(s4 != 0)
    { *(pT + (s4-1)*neqf + (u4-1)) = -1; }
}

if( *(ptrans + i) == 4 ){
    s1 = *(pjcode + (node1-1)*3 + 2); if(s1 != 0)
    { *(pT + (s1-1)*neqf + (u1-1)) = 1; }
    s2 = *(pjcode + (node2-1)*3 + 2); if(s2 != 0)
    { *(pT + (s2-1)*neqf + (u2-1)) = 1; }
    s3 = *(pjcode + (node3-1)*3 + 2); if(s3 != 0)
    { *(pT + (s3-1)*neqf + (u3-1)) = 1; }
    s4 = *(pjcode + (node4-1)*3 + 2); if(s4 != 0)
    { *(pT + (s4-1)*neqf + (u4-1)) = 1; }
}

```

```

    }
    if( *(ptrans + i) == 5 ){
        s1 = *(pjcode + (node1-1)*3 + 0); if(s1 != 0)
        { *(pT + (s1-1)*neqf + (u1-1)) = -1; }
        s2 = *(pjcode + (node2-1)*3 + 0); if(s2 != 0)
        { *(pT + (s2-1)*neqf + (u2-1)) = -1; }
        s3 = *(pjcode + (node3-1)*3 + 0); if(s3 != 0)
        { *(pT + (s3-1)*neqf + (u3-1)) = -1; }
        s4 = *(pjcode + (node4-1)*3 + 0); if(s4 != 0)
        { *(pT + (s4-1)*neqf + (u4-1)) = -1; }
    }
    if( *(ptrans + i) == 6 ){
        s1 = *(pjcode + (node1-1)*3 + 1); if(s1 != 0)
        { *(pT + (s1-1)*neqf + (u1-1)) = -1; }
        s2 = *(pjcode + (node2-1)*3 + 1); if(s2 != 0)
        { *(pT + (s2-1)*neqf + (u2-1)) = -1; }
        s3 = *(pjcode + (node3-1)*3 + 1); if(s3 != 0)
        { *(pT + (s3-1)*neqf + (u3-1)) = -1; }
        s4 = *(pjcode + (node4-1)*3 + 1); if(s4 != 0)
        { *(pT + (s4-1)*neqf + (u4-1)) = -1; }
    }

    // multiply the transformation T matrix with the area
    // A matrix
    for(r = 0; r <= neqs-1; ++r){
        for(t = 0; t <= neqf-1; ++t){
            sum = 0;
            for(s = 0; s <= neqf-1; ++s){
                sum += *(pT + r*neqf + s) *
                    (*(pA + s*neqf + t));
            }
            *(pL + (r*neqf) + t) += sum;
        }
    }

    // Zero out A matrix for this interface
    for(r = 0; r <= neqf - 1; ++r){
        for(t = 0; t <= neqf - 1; ++t){
            *(pA + r*neqf + t) = 0;
        }
    }

} // End interface loop
} // End If statement

```

```

    printf("Exiting Prop Function\n");

}

/* This function reads in the joint number, jnum, the joint
   direction, jdir, and the applied force, force */
int load (double *pq, long int *pjcode)
{
    printf("Entering Load Function\n");
    //    Initialize function variables
    long int i, k, jnum, jdir;
    double force;

    //    Zero-out generalized load vector
    for (i = 0; i <= neq - 1; ++i) {
        *(pq+i) = 0;
    }
    fscanf(ifp, "%ld\t%ld\t%lf\n", &jnum, &jdir, &force);
    if (jnum != 0) { // Check for joint loading
        while (jnum != 0) { // Check for last joint load
            k = *(pjcode+(jnum-1)*3+(jdir-1));
            // Scan and load jcode
        //    Store only joint loads corresponding to active global DOFs
            switch (k) {
                case (0):
                    break; // Do not store loads at supports
                default:
                    *(pq+k-1) = force;
                    break;
            }
            fscanf(ifp, "%ld\t%ld\t%lf\n", &jnum, &jdir, &force);
        }
    } else {
        fprintf(ofp, "\n***ERROR*** No joint forces "
                "present in input file");
        return 1;
    }
    return 0;

    printf("Exiting Load Function\n");
}

```

```

// This function calculates the internal force vector at time t,
// given the corresponding displacement vector, d
void multiplic (double *pmatrix, double *parray, double *presult,
               long int eqns)
{
    // Initialize function variables
    long int i, j, k;
    double sum;

    for( k = 0; k <= eqns - 1; ++k ){
        sum = 0;
        for( i = 0; i <= eqns - 1; ++i ){
            sum += *(pmatrix + k*eqns + i) * (*(parray + i));
        }
        *(presult + k) = sum;
    }
}

// This function computes the mass-proportional damping array
void damping (double *psd, double damp, double *psm1, double speed,
              long int *pelvn, long int *pbnodes, long int *pelnumf,
              double *pA, double *pbA, long int *pjcodef)
{
    // Initialize function variables
    int i, j, k, bnode, elm_num, bdof;
    double alpha1, alpha2, sum, area;

    // Initialize damping array to zero
    for (i = 0; i <= (neq*neq) - 1; ++i){
        *(psd + i) = 0;
    }

    // Initialize areas matrix to zero
    for(i = 0; i <= neqf-1; ++i){
        for(j = 0; j <= neqf-1; ++j){
            *(pbA+i*neqf+j)=0;
        }
    }

    // Define damping parameters
    alpha1 = damp;
    alpha2 = (2093.7319)/speed;

    // Compute and assign boundary node areas
    for ( i = 0; i <= neqf-1; ++i){

```

```

        for ( j = 0; j <= neqf-1; ++j){
            if( *(pA + i*neqf + j) != 0 ){
                area = *(pA + i*neqf + j);
            }
        }
    }
    //printf("Area = %lf\n", area);
    for (i = 0; i <= nbc-1; ++i){
        bnode = *(pbnodes + i);
        bdof = *(pjcodef + (bnode - 1) * 3);
        for (j = 0; j <= nef-1; ++j){
            elm_num = *(pelnumf + j)) - 1;
            for (k = 0; k <= 7; ++k){
                if ( *(pelvn + elm_num*8 + k) == bnode ){
                    *(pbA + (bdof-neqs-1)*neqf + (bdof-neqs-1)) =
                        *(pbA + (bdof-neqs-1)*neqf +
                            (bdof-neqs-1)) + (area/4.0);
                }
            }
        }
    }
}

// Compute damping array
for(i = 0; i <= neqs-1; ++i){
    for(j = 0; j <= neqs-1; ++j){
        *(psd + i*neq + j) = alpha1 *
            (*(psm1 + i*neqs + j));
    }
}
for(i = 0; i <= neqf-1; ++i){
    for(j = 0; j <= neqf-1; ++j){
        *(psd + ((i+neqs)*neq) + (j+neqs))
            = alpha2 * (*(pbA + i*neqf + j));
    }
}
printf("Exiting Damping Function\n");
}

// This function computes the system mass matrix and stores
// it as an array
void mass1 (double *psmx, double *pro, double *px, long int *pelvn,
            long int *pelnumx, long int *pmcode, long int *plss,
            long int nelms, long int eqns)
{

```

```

// Initialize function variables
int i, j, k, ie, je, n, elm_num;
double sum;
double M[24][24];
// Total element mass matrix in global coord. system

// Initialize structure mass matrix vector to zero
for (i = 0; i <= (*plss) - 1; ++i){
    *(psmx+i) = 0;    // mass matrix array
}

for (n = 0; n <= nelms - 1; ++n) {
    // Initialize all elements to zero
    for (i = 0; i <= 23; ++i)
    {
        for (j = 0; j <= 23; ++j)
        {
            M[i][j] = 0;
        }
    }

    // Pass control to masse function
    elm_num = *(pelnumx + n) - 1;
    masse (&M[0][0], *(pro + n), px, pelvn, elm_num);

    // Sum all row elements of the mass matrix to the diagonal
    for(i = 0; i <= 23; ++i){
        sum = 0;
        for(j=0; j<=23; ++j){
            sum += M[i][j];
            M[i][j] = 0;
        }
        M[i][i] = sum;
    }

    // Assign to correct global locations.
    for (ie = 0; ie <= 23; ++ie) {
        i = *(pmcode+n*24+ie);
        if (i != 0) {
            for (je = 0; je <= 23; ++je) {
                j = *(pmcode+n*24+je);
                if (j != 0) {
                    if(eqns == neqs && nelms == ne)
                        {k = ((i-1) * eqns) + (j-1);}
                }
            }
        }
    }
}

```



```

    }
    for (i = 0; i <= 23; ++i) {
        for (j = 0; j <= 23; ++j) {
            N_N[i][j] = 0;
        }
    }

    // Define current integration point
    intpt = c + 1;

    // Compute the Jacobian matrix, it's determinant, and inverse
    jacobian(px, pelvn, n, intpt, &J[0][0], &J_inv[0][0], &detJ);

    // Compute the shape function matrix
    for (i = 0; i <= 7; ++i){
        for (j = 0; j <= 2; ++j){
            N[j][i*3+j] = shape(i+1,0,intpt);
        }
    }

    // Compute M_integration_point = ro * transpose(N) * N * detJ
    for (i = 0; i <= 23; ++i) {
        for (j = 0; j <= 23; ++j) {
            sum = 0;
            for (k = 0; k <= 2; ++k) {
                sum += ro * N[k][i] * N[k][j] * detJ;
            }
            *(pmt+i*24+j) += sum;
        }
    }
} // end loop over integration points
}

void mass2 (double *psmx, double *pro, double *px, long int *pelvn,
            long int *pmcode, double *pc1, double *pc2, double *pc3,
            long int nelms, long int eqns, long int *plss)
{
    // Initialize function variables
    int i, j, k, n;
    double sum, vol;
    double M[24][24];
    // Total element mass matrix in global coord. system
    double m[24];
    // Array of local lumped mass matrix diagonal elements

```

```

// Initialize structure mass matrix vector to zero
for (i = 0; i <= (*plss) - 1; ++i){
    *(psmx+i) = 0;    // mass matrix array
}

for (n = 0; n <= nelms - 1; ++n) {
    // Initialize all elements to zero
    for (i = 0; i <= 23; ++i)
    {
        for (j = 0; j <= 23; ++j)
        {
            M[i][j] = 0;
        }
        m[i] = 0;
    }

    vol = volume(pelvn, px, n);

    // Compute element mass matrix
    for(i = 0; i <= 23; ++i){
        M[i][i] = (*(pro+n)) * (vol/8);
    }

    // Sum all row elements of mass matrix to the diagonal array
    for(i = 0; i <= 23; ++i){
        sum = 0;
        for(j=0; j<=23; ++j){
            sum += M[i][j];
        }
        m[i] = sum;
    }

    // Assign the diagonal values to the correct global location
    for (i = 0; i <= 23; ++i) {
        j = *(pmcode+n*24+i);
        if (j != 0) {
            if(eqns == neqs && nelms == ne)
                {k = ((j-1) * eqns) + (j-1);}
            else if(eqns == neqf && nelms == nef)
                {k = ((j-neqs-1) * eqns) + (j-neqs-1);}
            *(psmx + k) += m[i];
        }
    }
}

```

```

        // End element loop
    }
}

// Consistent mass matrix and global matrix.
void mass3 (double *psmx, double *pro, double *px, long int *pelvn,
            long int *pelnumx, long int *pmcode, long int *plss,
            long int nelms, long int eqns)
{
    // Initialize function variables
    int i, j, k, ie, je, n, elm_num;
    double sum;
    double M[24][24];
    // Total element mass matrix in global coord. system

    // Initialize structure mass matrix vector to zero
    for (i = 0; i <= (*plss) - 1; ++i){
        *(psmx+i) = 0;    // mass matrix array
    }

    for (n = 0; n <= nelms - 1; ++n) {
        // Initialize all elements to zero
        for (i = 0; i <= 23; ++i)
        {
            for (j = 0; j <= 23; ++j)
            {
                M[i][j] = 0;
            }
        }

        // Pass control to masse function
        elm_num = *(pelnumx + n) - 1;
        masse (&M[0][0], *(pro + n), px, pelvn, elm_num);

        // Assign to correct global locations.
        for (ie = 0; ie <= 23; ++ie) {
            i = *(pmcode+n*24+ie);
            if (i != 0) {
                for (je = 0; je <= 23; ++je) {
                    j = *(pmcode+n*24+je);
                    if (j != 0) {
                        if(eqns == neqs && nelms == ne)
                            {k = ((i-1) * eqns) + (j-1);}
                    }
                }
            }
        }
    }
}

```



```

int i, j, k, c, intpt;
int p, q, r;
double alpha;
double sum, detJ;
double J[3][3], J_inv[3][3]; // Jacobian matrix
double B[6][24]; // Membrane strain-displacement matrix
double C[6][6]; // Plane stress constitutive matrix
double B_C[24][6];
double B_C_B[24][24];
double temp[3];

// Initialize matrices to zero
for (i = 0; i <= 5; ++i) {
    for (j = 0; j <= 5; ++j) {
        C[i][j] = 0;
    }
}
for (i = 0; i <= 23; ++i) {
    for (j = 0; j <= 23; ++j) {
        *(pK+i*24+j) = 0;
    }
}

// Define constitutive matrix w/ nu and E of this element
alpha = ( E*(1-nu) )/( (1+nu)*(1-(2*nu)) );
C[0][0] = C[1][1] = C[2][2] = alpha;
C[0][1] = C[0][2] = C[1][2] = C[1][0] = C[2][0] = C[2][1] =
    (alpha*nu) / (1-nu);
C[3][3] = C[4][4] = C[5][5] = (alpha*(1-(2*nu)))/(2*(1-nu));

// For each integration point
for (c = 0; c <= 7; ++c)
{
    // Initialize required matrices to zero
    for (i = 0; i <= 5; ++i) {
        for (j = 0; j <= 23; ++j) {
            B[i][j] = 0;
            B_C[j][i] = 0;
        }
    }
    for (i = 0; i <= 23; ++i) {
        for (j = 0; j <= 23; ++j) {
            B_C_B[i][j] = 0;
        }
    }
}

```

```

    }

    // Define current integration point
    intpt = c + 1;

    // Compute the Jacobian matrix, it's determinant, and inverse
    jacobian(px, pelvn, n, intpt, &J[0][0], &J_inv[0][0], &detJ);

    // Compute the strain-disp matrix
    for (i = 0; i <= 7; ++i){
        temp[0] = temp[1] = temp[2] = 0;
        for (j = 0; j <= 2; ++j){
            temp[0] += J_inv[0][j]*shape(i+1,j+1,intpt);
            temp[1] += J_inv[1][j]*shape(i+1,j+1,intpt);
            temp[2] += J_inv[2][j]*shape(i+1,j+1,intpt);
        }
        B[0][0+(3*i)] = B[3][1+(3*i)] = B[5][2+(3*i)] = temp[0];
        B[1][1+(3*i)] = B[3][0+(3*i)] = B[4][2+(3*i)] = temp[1];
        B[2][2+(3*i)] = B[4][1+(3*i)] = B[5][0+(3*i)] = temp[2];
    }

    // Compute K_integration_point = transpose(B) * C * B * detJ
    for (p = 0; p<=23; ++p) {
        for (q = 0; q<=5; ++q) {
            sum = 0;
            for (r = 0; r<= 5; ++r) {
                sum += B[r][p] * C[r][q];
            }
            B_C[p][q] = sum;
        }
    }

    for (i = 0; i <= 23; ++i) {
        for (j = 0; j <= 23; ++j) {
            sum = 0;
            for (k = 0; k <= 5; ++k) {
                sum += B_C[i][k] * B[k][j] * detJ;
            }
            *(pK+i*24+j) += sum;
        }
    }
} // end loop over integration points
}

```



```

// solves the system Ax=y using LU factorization
int solve(double *py,double *pA,double *px,int *pipiv)
{
    int i, j; // Counter variables
    time_t start,end;
    long double dif;
    int info;

    for(i = 0; i <= neq-1; ++i){
*(px + i) = *(py + i);
    }

    // Run solver
    time (&start);
    info = clapack_dgesv(CblasRowMajor, neq, 1, pA, neq, pipiv,
        px, neq);
    if (info != 0){
        fprintf(stderr, "failure with error %d\n", info);
        return 1;
    }
    time (&end);
    dif = difftime (end,start);
    printf("System solved in %llf seconds.\n", dif);

    return 0;
}

void jacobian (double *px, long int *pelvn, int n, int intpt,
    double *pJ, double *pJ_inv, double *pdetJ)
{
    // Initialize function variables
    int i, j, k;
    int nodes[8];
    double x1, x2, x3, x4, x5, x6, x7, x8;
    double y1, y2, y3, y4, y5, y6, y7, y8;
    double z1, z2, z3, z4, z5, z6, z7, z8;

    // These are the global coordinates of the current element
    for( i = 0; i <= 7; ++i){
        nodes[i] = *(pelvn+n*8+i) - 1;
    }
    x1 = *(px+nodes[0]*3+0); y1 = *(px+nodes[0]*3+1);
    z1 = *(px+nodes[0]*3+2);
    x2 = *(px+nodes[1]*3+0); y2 = *(px+nodes[1]*3+1);

```

```

    z2 = *(px+nodes[1]*3+2);
x3 = *(px+nodes[2]*3+0); y3 = *(px+nodes[2]*3+1);
    z3 = *(px+nodes[2]*3+2);
x4 = *(px+nodes[3]*3+0); y4 = *(px+nodes[3]*3+1);
    z4 = *(px+nodes[3]*3+2);
x5 = *(px+nodes[4]*3+0); y5 = *(px+nodes[4]*3+1);
    z5 = *(px+nodes[4]*3+2);
x6 = *(px+nodes[5]*3+0); y6 = *(px+nodes[5]*3+1);
    z6 = *(px+nodes[5]*3+2);
x7 = *(px+nodes[6]*3+0); y7 = *(px+nodes[6]*3+1);
    z7 = *(px+nodes[6]*3+2);
x8 = *(px+nodes[7]*3+0); y8 = *(px+nodes[7]*3+1);
    z8 = *(px+nodes[7]*3+2);

/* shape function is called as: shape(global coord. dir.,
local coord. dir, integration point)*/
*(pJ+0*3+0) = (shape(1,1,intpt)*x1)+(shape(2,1,intpt)*x2)+
    (shape(3,1,intpt)*x3)+(shape(4,1,intpt)*x4)+
    (shape(5,1,intpt)*x5)+(shape(6,1,intpt)*x6)+
    (shape(7,1,intpt)*x7)+(shape(8,1,intpt)*x8);
*(pJ+0*3+1) = (shape(1,1,intpt)*y1)+(shape(2,1,intpt)*y2)+
    (shape(3,1,intpt)*y3)+(shape(4,1,intpt)*y4)+
    (shape(5,1,intpt)*y5)+(shape(6,1,intpt)*y6)+
    (shape(7,1,intpt)*y7)+(shape(8,1,intpt)*y8);
*(pJ+0*3+2) = (shape(1,1,intpt)*z1)+(shape(2,1,intpt)*z2)+
    (shape(3,1,intpt)*z3)+(shape(4,1,intpt)*z4)+
    (shape(5,1,intpt)*z5)+(shape(6,1,intpt)*z6)+
    (shape(7,1,intpt)*z7)+(shape(8,1,intpt)*z8);
*(pJ+1*3+0) = (shape(1,2,intpt)*x1)+(shape(2,2,intpt)*x2)+
    (shape(3,2,intpt)*x3)+(shape(4,2,intpt)*x4)+
    (shape(5,2,intpt)*x5)+(shape(6,2,intpt)*x6)+
    (shape(7,2,intpt)*x7)+(shape(8,2,intpt)*x8);
*(pJ+1*3+1) = (shape(1,2,intpt)*y1)+(shape(2,2,intpt)*y2)+
    (shape(3,2,intpt)*y3)+(shape(4,2,intpt)*y4)+
    (shape(5,2,intpt)*y5)+(shape(6,2,intpt)*y6)+
    (shape(7,2,intpt)*y7)+(shape(8,2,intpt)*y8);
*(pJ+1*3+2) = (shape(1,2,intpt)*z1)+(shape(2,2,intpt)*z2)+
    (shape(3,2,intpt)*z3)+(shape(4,2,intpt)*z4)+
    (shape(5,2,intpt)*z5)+(shape(6,2,intpt)*z6)+
    (shape(7,2,intpt)*z7)+(shape(8,2,intpt)*z8);
*(pJ+2*3+0) = (shape(1,3,intpt)*x1)+(shape(2,3,intpt)*x2)+
    (shape(3,3,intpt)*x3)+(shape(4,3,intpt)*x4)+
    (shape(5,3,intpt)*x5)+(shape(6,3,intpt)*x6)+
    (shape(7,3,intpt)*x7)+(shape(8,3,intpt)*x8);

```

```

*(pJ+2*3+1) = (shape(1,3,intpt)*y1)+(shape(2,3,intpt)*y2)+
              (shape(3,3,intpt)*y3)+(shape(4,3,intpt)*y4)+
              (shape(5,3,intpt)*y5)+(shape(6,3,intpt)*y6)+
              (shape(7,3,intpt)*y7)+(shape(8,3,intpt)*y8);
*(pJ+2*3+2) = (shape(1,3,intpt)*z1)+(shape(2,3,intpt)*z2)+
              (shape(3,3,intpt)*z3)+(shape(4,3,intpt)*z4)+
              (shape(5,3,intpt)*z5)+(shape(6,3,intpt)*z6)+
              (shape(7,3,intpt)*z7)+(shape(8,3,intpt)*z8);

*pdetJ = (((*(pJ+0*3+0))*(*(pJ+1*3+1))*(*(pJ+2*3+2)))+
          (((*(pJ+0*3+1))*(*(pJ+1*3+2))*(*(pJ+2*3+0)))+
          (((*(pJ+0*3+2))*(*(pJ+1*3+0))*(*(pJ+2*3+1))))-
          (((*(pJ+2*3+0))*(*(pJ+1*3+1))*(*(pJ+0*3+2)))+
          (((*(pJ+2*3+1))*(*(pJ+1*3+2))*(*(pJ+0*3+0)))+
          (((*(pJ+2*3+2))*(*(pJ+0*3+1))*(*(pJ+1*3+0)))));

// Compute the inverse of the jacobian matrix
// The adjoint matrix of the transpose of J * (1/detJ)
*(pJ_inv+0*3+0)=(((*(pJ+1*3+1))*(*(pJ+2*3+2))-
                 (((*(pJ+1*3+2))*(*(pJ+2*3+1))))*(1.0/(*(pdetJ))));
*(pJ_inv+0*3+1)=((-1)*(((*(pJ+0*3+1))*(*(pJ+2*3+2))-
                        (((*(pJ+0*3+2))*(*(pJ+2*3+1))))*(1.0/(*(pdetJ))));
*(pJ_inv+0*3+2)=(((*(pJ+0*3+1))*(*(pJ+1*3+2))-
                 (((*(pJ+0*3+2))*(*(pJ+1*3+1))))*(1.0/(*(pdetJ))));
*(pJ_inv+1*3+0)=((-1)*(((*(pJ+1*3+0))*(*(pJ+2*3+2))-
                        (((*(pJ+1*3+2))*(*(pJ+2*3+0))))*(1.0/(*(pdetJ))));
*(pJ_inv+1*3+1)=(((*(pJ+0*3+0))*(*(pJ+2*3+2))-
                 (((*(pJ+0*3+2))*(*(pJ+2*3+0))))*(1.0/(*(pdetJ))));
*(pJ_inv+1*3+2)=((-1)*(((*(pJ+0*3+0))*(*(pJ+1*3+2))-
                        (((*(pJ+0*3+2))*(*(pJ+1*3+0))))*(1.0/(*(pdetJ))));
*(pJ_inv+2*3+0)=(((*(pJ+1*3+0))*(*(pJ+2*3+1))-
                 (((*(pJ+1*3+1))*(*(pJ+2*3+0))))*(1.0/(*(pdetJ))));
*(pJ_inv+2*3+1)=((-1)*(((*(pJ+0*3+0))*(*(pJ+2*3+1))-
                        (((*(pJ+0*3+1))*(*(pJ+2*3+0))))*(1.0/(*(pdetJ))));
*(pJ_inv+2*3+2)=(((*(pJ+0*3+0))*(*(pJ+1*3+1))-
                 (((*(pJ+0*3+1))*(*(pJ+1*3+0))))*(1.0/(*(pdetJ))));

}

/* This function transforms the element stiffness matrix from
local into global coordinate system */
void transform (double *pkt, double *pT, double *pK, int n)
{
    // Initialize function variables

```

```

int i, j, k;
double temp[n][n];
double sum;

// Multiply transpose of transformation matrix by
// element stiffness matrix
for (i = 0; i <= n - 1; i++) {
    for (j = 0; j <= n - 1; j++) {
        sum = 0;
        for (k = 0; k <= n - 1; k++) {
            sum += (*(pT+k*n+i)) * (*(pkt+k*n+j));
        }
        temp[i][j] = sum;
    }
}

// Multiply above result by transformation matrix
for (i = 0; i <= n - 1; i++) {
    for (j = 0; j <= n - 1; j++) {
        sum = 0;
        for (k = 0; k <= n - 1; k++) {
            sum += temp[i][k] * (*(pT+k*n+j));
        }
        *(pK+i*n+j) = sum;
    }
}

printf("Exiting Transform Function\n");
}

// This function returns shape function derivatives
double shape(int n, int x, int intpt)
{
    // Initialize function variables
    double r, s, t; // integration point coordinates

    // Assign Integration Point Coordinates
    r = s = t = 1/sqrt(3.0);
    switch (intpt) {
        case 1:
            r = -1*r; s = -1*s; t = -1*t;
            break;
        case 2:
            r = 1*r; s = -1*s; t = -1*t;

```

```

        break;
case 3:
    r = 1*r; s = -1*s; t = 1*t;
    break;
case 4:
    r = -1*r; s = -1*s; t = 1*t;
    break;
case 5:
    r = -1*r; s = 1*s; t = -1*t;
    break;
case 6:
    r = 1*r; s = 1*s; t = -1*t;
    break;
case 7:
    r = 1*r; s = 1*s; t = 1*t;
    break;
case 8:
    r = -1*r; s = 1*s; t = 1*t;
    break;
default:
    fprintf(ofp, "\n***ERROR*** Integration Point Value");
    return 1;
    break;
}

```

```

// Determine Shape Function Derivative Value
if (x == 0){ // zero means no derivative
    switch (n) {
        case 1:
            return (1/8.0)*(1-r)*(1-s)*(1-t);
            break;
        case 2:
            return (1/8.0)*(1+r)*(1-s)*(1-t);
            break;
        case 3:
            return (1/8.0)*(1+r)*(1-s)*(1+t);
            break;
        case 4:
            return (1/8.0)*(1-r)*(1-s)*(1+t);
            break;
        case 5:
            return (1/8.0)*(1-r)*(1+s)*(1-t);
            break;
        case 6:

```

```

        return (1/8.0)*(1+r)*(1+s)*(1-t);
        break;
    case 7:
        return (1/8.0)*(1+r)*(1+s)*(1+t);
        break;
    case 8:
        return (1/8.0)*(1-r)*(1+s)*(1+t);
        break;
    default:
        fprintf(ofp, "\n***ERROR*** Shape Function");
        return 1;
        break;
    }
}

if (x == 1){
    switch (n) {
        case 1:
            return (-1/8.0)*(1-s)*(1-t);
            break;
        case 2:
            return (1/8.0)*(1-s)*(1-t);
            break;
        case 3:
            return (1/8.0)*(1-s)*(1+t);
            break;
        case 4:
            return (-1/8.0)*(1-s)*(1+t);
            break;
        case 5:
            return (-1/8.0)*(1+s)*(1-t);
            break;
        case 6:
            return (1/8.0)*(1+s)*(1-t);
            break;
        case 7:
            return (1/8.0)*(1+s)*(1+t);
            break;
        case 8:
            return (-1/8.0)*(1+s)*(1+t);
            break;
        default:
            fprintf(ofp, "\n***ERROR*** Shape Function");
            return 1;
    }
}

```

```

        break;
    }
}

if (x == 2){
    switch (n) {
        case 1:
            return  $(-1/8.0) * (1-r) * (1-t)$ ;
            break;
        case 2:
            return  $(-1/8.0) * (1+r) * (1-t)$ ;
            break;
        case 3:
            return  $(-1/8.0) * (1+r) * (1+t)$ ;
            break;
        case 4:
            return  $(-1/8.0) * (1-r) * (1+t)$ ;
            break;
        case 5:
            return  $(1/8.0) * (1-r) * (1-t)$ ;
            break;
        case 6:
            return  $(1/8.0) * (1+r) * (1-t)$ ;
            break;
        case 7:
            return  $(1/8.0) * (1+r) * (1+t)$ ;
            break;
        case 8:
            return  $(1/8.0) * (1-r) * (1+t)$ ;
            break;
        default:
            fprintf(ofp, "\n***ERROR*** Shape Function");
            return 1;
            break;
    }
}

if (x == 3){
    switch (n) {
        case 1:
            return  $(-1/8.0) * (1-r) * (1-s)$ ;
            break;
        case 2:
            return  $(-1/8.0) * (1+r) * (1-s)$ ;

```

```

        break;
    case 3:
        return (1/8.0)*(1+r)*(1-s);
        break;
    case 4:
        return (1/8.0)*(1-r)*(1-s);
        break;
    case 5:
        return (-1/8.0)*(1-r)*(1+s);
        break;
    case 6:
        return (-1/8.0)*(1+r)*(1+s);
        break;
    case 7:
        return (1/8.0)*(1+r)*(1+s);
        break;
    case 8:
        return (1/8.0)*(1-r)*(1+s);
        break;
    default:
        fprintf(ofp, "\n***ERROR*** Shape Function");
        return 1;
        break;
    }
}

else {
    fprintf(ofp, "\n***ERROR*** Shape Function Derivative");
    return 1;
}

}

// This function computes the volume of the cube
double volume(long int *pelvn, double *px, int n)
{
    int i, j, k;
    int nodes[8]; //nodes
    double x1[3], x2[3], x3[3], x4[3], x5[3], x6[3], x7[3], x8[3];
    // node coord.
    double xe[3], xw[3], xn[3], xs[3], xt[3], xb[3];
    //center point coord.
    double sum;

```



```

// These are the global coordinates of the current element
for( i = 0; i <= 7; ++i){
    nodes[i] = *(pelvn+n*8+i) - 1;
}
x1[0] = *(px+nodes[0]*3+0); x1[1] = *(px+nodes[0]*3+1); x1[2]
    = *(px+nodes[0]*3+2);
x2[0] = *(px+nodes[1]*3+0); x2[1] = *(px+nodes[1]*3+1); x2[2]
    = *(px+nodes[1]*3+2);
x3[0] = *(px+nodes[2]*3+0); x3[1] = *(px+nodes[2]*3+1); x3[2]
    = *(px+nodes[2]*3+2);
x4[0] = *(px+nodes[3]*3+0); x4[1] = *(px+nodes[3]*3+1); x4[2]
    = *(px+nodes[3]*3+2);
x5[0] = *(px+nodes[4]*3+0); x5[1] = *(px+nodes[4]*3+1); x5[2]
    = *(px+nodes[4]*3+2);
x6[0] = *(px+nodes[5]*3+0); x6[1] = *(px+nodes[5]*3+1); x6[2]
    = *(px+nodes[5]*3+2);
x7[0] = *(px+nodes[6]*3+0); x7[1] = *(px+nodes[6]*3+1); x7[2]
    = *(px+nodes[6]*3+2);
x8[0] = *(px+nodes[7]*3+0); x8[1] = *(px+nodes[7]*3+1); x8[2]
    = *(px+nodes[7]*3+2);

// Compute center points of each face
for(i = 0; i <= 2; ++i){
    xe[i] = (x2[i] + x3[i] + x6[i] + x7[i])/4;
    xw[i] = (x1[i] + x4[i] + x5[i] + x8[i])/4;
    xn[i] = (x5[i] + x6[i] + x7[i] + x8[i])/4;
    xs[i] = (x1[i] + x2[i] + x3[i] + x4[i])/4;
    xt[i] = (x3[i] + x4[i] + x7[i] + x8[i])/4;
    xb[i] = (x1[i] + x2[i] + x5[i] + x6[i])/4;
}

sum = 0;
// triangle 1: x1, x2, xs
sum += (x1[2] + x2[2] + xs[2])*((x2[0] - x1[0])*(xs[1] - x1[1]) -
    (xs[0] - x1[0])*(x2[1] - x1[1]));
// triangle 2: x2, x3, xs
sum += (x2[2] + x3[2] + xs[2])*((x3[0] - x2[0])*(xs[1] - x2[1]) -
    (xs[0] - x2[0])*(x3[1] - x2[1]));
// triangle 3: x3, x4, xs
sum += (x3[2] + x4[2] + xs[2])*((x4[0] - x3[0])*(xs[1] - x3[1]) -
    (xs[0] - x3[0])*(x4[1] - x3[1]));
// triangle 4: x4, x1, xs
sum += (x4[2] + x1[2] + xs[2])*((x1[0] - x4[0])*(xs[1] - x4[1]) -
    (xs[0] - x4[0])*(x1[1] - x4[1]));

```

```

// triangle 5: x2, x6, xe
sum += (x2[2] + x6[2] + xe[2])*((x6[0] - x2[0])*(xe[1] - x2[1]) -
    (xe[0] - x2[0])*(x6[1] - x2[1]));
// triangle 6: x6, x7, xe
sum += (x6[2] + x7[2] + xe[2])*((x7[0] - x6[0])*(xe[1] - x6[1]) -
    (xe[0] - x6[0])*(x7[1] - x6[1]));
// triangle 7: x7, x3, xe
sum += (x7[2] + x3[2] + xe[2])*((x3[0] - x7[0])*(xe[1] - x7[1]) -
    (xe[0] - x7[0])*(x3[1] - x7[1]));
// triangle 8: x3, x2, xe
sum += (x3[2] + x2[2] + xe[2])*((x2[0] - x3[0])*(xe[1] - x3[1]) -
    (xe[0] - x3[0])*(x2[1] - x3[1]));
// triangle 9: x4, x3, xt
sum += (x4[2] + x3[2] + xt[2])*((x3[0] - x4[0])*(xt[1] - x4[1]) -
    (xt[0] - x4[0])*(x3[1] - x4[1]));
// triangle 10: x3, x7, xt
sum += (x3[2] + x7[2] + xt[2])*((x7[0] - x3[0])*(xt[1] - x3[1]) -
    (xt[0] - x3[0])*(x7[1] - x3[1]));
// triangle 11: x7, x8, xt
sum += (x7[2] + x8[2] + xt[2])*((x8[0] - x7[0])*(xt[1] - x7[1]) -
    (xt[0] - x7[0])*(x8[1] - x7[1]));
// triangle 12: x8, x4, xt
sum += (x8[2] + x4[2] + xt[2])*((x4[0] - x8[0])*(xt[1] - x8[1]) -
    (xt[0] - x8[0])*(x4[1] - x8[1]));
// triangle 13: x2, x1, xb
sum += (x2[2] + x1[2] + xb[2])*((x1[0] - x2[0])*(xb[1] - x2[1]) -
    (xb[0] - x2[0])*(x1[1] - x2[1]));
// triangle 14: x6, x2, xb
sum += (x6[2] + x2[2] + xb[2])*((x2[0] - x6[0])*(xb[1] - x6[1]) -
    (xb[0] - x6[0])*(x2[1] - x6[1]));
// triangle 15: x5, x6, xb
sum += (x5[2] + x6[2] + xb[2])*((x6[0] - x5[0])*(xb[1] - x5[1]) -
    (xb[0] - x5[0])*(x6[1] - x5[1]));
// triangle 16: x1, x5, xb
sum += (x1[2] + x5[2] + xb[2])*((x5[0] - x1[0])*(xb[1] - x1[1]) -
    (xb[0] - x1[0])*(x5[1] - x1[1]));
// triangle 17: x1, x4, xw
sum += (x1[2] + x4[2] + xw[2])*((x4[0] - x1[0])*(xw[1] - x1[1]) -
    (xw[0] - x1[0])*(x4[1] - x1[1]));
// triangle 18: x5, x1, xw
sum += (x5[2] + x1[2] + xw[2])*((x1[0] - x5[0])*(xw[1] - x5[1]) -
    (xw[0] - x5[0])*(x1[1] - x5[1]));
// triangle 19: x8, x5, xw
sum += (x8[2] + x5[2] + xw[2])*((x5[0] - x8[0])*(xw[1] - x8[1]) -

```

```

        (xw[0] - x8[0])*(x5[1] - x8[1]));
// triangle 20: x4, x8, xw
sum += (x4[2] + x8[2] + xw[2])*((x8[0] - x4[0])*(xw[1] - x4[1]) -
    (xw[0] - x4[0])*(x8[1] - x4[1]));
// triangle 21: x6, x5, xn
sum += (x6[2] + x5[2] + xn[2])*((x5[0] - x6[0])*(xn[1] - x6[1]) -
    (xn[0] - x6[0])*(x5[1] - x6[1]));
// triangle 22: x5, x8, xn
sum += (x5[2] + x8[2] + xn[2])*((x8[0] - x5[0])*(xn[1] - x5[1]) -
    (xn[0] - x5[0])*(x8[1] - x5[1]));
// triangle 23: x8, x7, xn
sum += (x8[2] + x7[2] + xn[2])*((x7[0] - x8[0])*(xn[1] - x8[1]) -
    (xn[0] - x8[0])*(x7[1] - x8[1]));
// triangle 24: x7, x6, xn
sum += (x7[2] + x6[2] + xn[2])*((x6[0] - x7[0])*(xn[1] - x7[1]) -
    (xn[0] - x7[0])*(x6[1] - x7[1]));
return (sum/6.0);
}

```

```

// This function computes the dot product of two vectors
double dot (double *pa, double *pb, int n)
{
    // Initialize function variables
    int i;
    double dp = 0;

    for (i = 0; i <= n - 1; ++i) {
        dp += *(pa+i) * (*(pb+i));
    }
    return dp;
}

```

```

// This function computes the cross product of two vectors
void cross (double *pa, double *pb, double *pc, int flag)
{
    // Initialize function variables
    int i;
    double length;

    // Compute cross product
    *(pc) = (*(pa+1)) * (*(pb+2)) - (*(pa+2)) * (*(pb+1));
    *(pc+1) = (*(pa+2)) * (*(pb)) - (*(pa)) * (*(pb+2));
    *(pc+2) = (*(pa)) * (*(pb+1)) - (*(pa+1)) * (*(pb));
}

```

```

    if (flag == 1) {
        // Compute length and normalize cross product
        length = sqrt((*pc) * (*pc) + (*pc+1) * (*pc+1) +
            (*pc+2) * (*pc+2));
        for (i = 0; i <= 2; ++i) {
            *(pc+i) /= length;
        }
    }
}

// This function closes the input and output files
int closeio (int flag)
{
    if (flag == 0) {
        // Close the I/O
        if (fclose(ifp) != 0) {
            printf("***ERROR*** Unable to close the input file\n");
        } else {
            printf("Input file is closed\n");
        }
        if (fclose(ofp) != 0) {
            printf("***ERROR*** Unable to close the output file\n");
        } else {
            printf("Output file is closed\n");
        }
    } else {
        // Notify user that solution failed
        fprintf(ofp, "\n\nSolution failed\n");
        printf("Solution failed, see output file\n");

        // Close the I/O
        if (fclose(ifp) != 0) {
            printf("***ERROR*** Unable to close the input file\n");
        } else {
            printf("Input file is closed\n");
        }
        if (fclose(ofp) != 0) {
            printf("***ERROR*** Unable to close the output file\n");
        } else {
            printf("Output file is closed\n");
        }
    }
    getchar();
    return 0;
}

```

```

}

// This function allocates memory for an array of type double
double * alloc_dbl (long int size)
{
    double *a;
    a = (double *) malloc(size * sizeof(double));
    if (a == NULL) {
        fprintf(ofp, "\n***ERROR*** Unable to allocate memory");
        return NULL;
    }
    return a;
}

// This function allocates memory for an array of type long int
long int * alloc_int (long int size)
{
    long int *a;
    a = (long int *) malloc(size * sizeof(long int));
    if (a == NULL) {
        fprintf(ofp, "\n***ERROR*** Unable to allocate memory");
        return NULL;
    }
    return a;
}

// This function allocates memory for an array of type int
int * alloc_int2 (int size)
{
    int *a;
    a = (int *) malloc(size * sizeof(int));
    if (a == NULL) {
        fprintf(ofp, "\n***ERROR*** Unable to allocate memory");
        return NULL;
    }
    return a;
}

// This function frees all allocated memory
int free_all (double **pp2p2d, int nd, long int **pp2p2i, int ni,
              int **pp2p2i2, int ni2, int flag)
{
    // Initialize function variables
    int i;

```

```

// Free allocated memory for arrays of type double
for (i = 0; i < nd; ++i) {
    if (*(pp2p2d+i) != NULL) {
        free (*(pp2p2d+i));
        *(pp2p2d+i) = NULL;
    }
}

// Free allocated memory for arrays of type long int
for (i = 0; i < ni; ++i) {
    if (*(pp2p2i+i) != NULL) {
        free (*(pp2p2i+i));
        *(pp2p2i+i) = NULL;
    }
}

// Free allocated memory for arrays of type int
for (i = 0; i < ni2; ++i) {
    if (*(pp2p2i2+i) != NULL) {
        free (*(pp2p2i2+i));
        *(pp2p2i2+i) = NULL;
    }
}

return closeio(flag);
}

```

BIBLIOGRAPHY

- [1] ADINA R. D., Inc., 71 Elton Ave Watertown, MA 02472 USA. *Automatic Dynamic Incremental Nonlinear Analysis System 8.3 Release Notes.*, October 2005.
- [2] E. et. al. Anderson. *LAPACK Users' Guide Third Edition.* Society for Industrial and Applied Mathematics, August 1999.
- [3] KL. Bathe. *Finite Element Procedures.* Prentice-Hall, 2006.
- [4] KL. Bathe and E.L. Wilson. *Numerical methods in finite element analysis.* Prentice-Hall, 1976.
- [5] T. Belytschko, W.K. Liu, and B. Moran. *Nonlinear Finite Element for Continua and Structures.* Wiley, 2000.
- [6] J.C. Bringham, W. Aquino, F.G. Mitri, J.F. Greenleaf, and M. Fatemi. Inverse estimation of viscoelastic material properties for solids immersed in fluids using vibroacoustic techniques. *Journal of Applied Physics*, 101, 2007.
- [7] L.H. Chen and D.G. Schweikert. Sound radiation from an arbitrary body. *J. Acoust. Soc. Amer.*, 35(10):1626–1632, 1963.
- [8] A.K. Chopra. Earthquake behavior of reservoir-dam systems. In *A.S.C.E. National Meeting on Structural Engineering*, 1968.
- [9] A. Craggs. The transient response of a coupled plate-acoustic system using plate and acoustic finite elements. *J. Sound and Vibration*, 15(4):509–528, 1971.
- [10] G.C. Everstine. Structural analogies for scalar field problems. *Int. J. numer. Meth. Engng.*, 17:471–476, 1981.
- [11] G.C. Everstine. A symmetric potential formulation for fluid-structure interaction. *J. Sound and Vibration*, 79:157–160, 1981.
- [12] G.C. Everstine. Structural-acoustic finite element analysis, with application to scattering. In H. Kardestuncer, editor, *Proc. 6th Invitational Symposium on the Unification of Finite Elements, Finite Differences, and Calculus of Variations*, pages 101–122, 1982.

- [13] G.C. Everstine. Finite element formulations of structural acoustics problems. *Comp. and Structures*, 65(3):307–321, 1995.
- [14] G.C. Everstine and G.C. Gaunaurd. Acoustic scattering by two submerged elastic spherical shells: mutual validation of the normal mode series and b.e.m. solutions. In *Proc. ASME Noise Control and Acoustics Division*, volume 24, pages 67–73, 1997.
- [15] G.C. Everstine and F.M. Henderson. Coupled finite element/boundary element approach for fluid-structure interaction. *J. Acoust. Soc. Amer.*, 87(5):1938–1947, 1990.
- [16] M.A. Hamdi and Y. Ousset. A displacement method for the analysis of vibrations of coupled fluid-structure systems. *Int. J. Num. Meth. in Engrg.*, 13(1):139–150, 1978.
- [17] J.K. Harbaugh, D.D. Sternlicht, A. Putney, M.D. Tinkle, and E.M. Lavelly. Acoustic modeling for sea-mine cad/cac development. *Oceans*, 1:850–854, 2005.
- [18] J.T. Hunt, M.R. Knittel, and D. Barach. Finite element approach to acoustic radiation from elastic structures. *J. Acoust. Soc. Amer.*, 55:269–280, 1974.
- [19] A.J. Kalinowski and C.W. Nebelung. Media-structure interaction computations employing frequency-dependent mesh size with the finite element method. *Shock Vib. Bull.*, 51(1):173–193, 1981.
- [20] J.B. Keller and D. Givoli. Exact non-reflecting boundary conditions. *J. Comput. Phys.*, 82:172–192, 1989.
- [21] E.T. Moyer. Performance of mapped infinite elements for exterior wave scattering applications. *Commun. appl. numer. Meth.*, 67:27–39, 1980.
- [22] R.E. Newton. Finite element study of shock induced cavitation. *American Society of Civil Engineers*, pages 90–110, 1980.
- [23] M.L. Palmeri, A.C. Sharma, R.R. Bouchard, R.W. Nightingale, and K.R. Nightingale. A finite-element method model of soft tissue response to impulsive acoustic radiation force. *IEEE Trans. on Ultrasonics, Ferroel., and Freq. Control.*, 52:1699–1712, 2005.

- [24] P.M. Pinsky and N.N. Abboud. Transient finite element analysis of the exterior structural acoustics problem. *Numerical Techniques in Acoustic Radiation*, 6:35–47, 1989.
- [25] L.L. Thompson. A review of finite element methods for time-harmonic acoustics. *J. Acoust. Soc. Am.*, 119(3):1315–1330, 2005.
- [26] M.F. Werby and G.J. Tango. Application of the extended boundary condition equations to scattering from fluid-loading objects. *Eng. Anal.*, pages 12–20, 1988.
- [27] D.T. Wilton. Acoustic radiation and scattering from elastic structures. *Int. J. Num. Meth. in Engrg.*, 13:123–138, 1978.
- [28] O.C. Zienkiewicz. Achievements and some unsolved problems of the finite element method. *Int. J. Num. Meth. Eng.*, 47:9–28, 2000.
- [29] O.C. Zienkiewicz and P. Bettles. Fluid-structure dynamic interaction and wave forces; an introduction to numerical treatment. *Int. J. for Num. Meth. in Eng.*, 13:1–16, 1978.
- [30] O.C. Zienkiewicz and Y. K. Cheung. *The Finite Element Method in Structural and Continuum Mechanics*. McGraw-Hill, 1967.
- [31] O.C. Zienkiewicz, B. Irons, and B. Nath. Natural frequencies of complex free and submerged structures by the finite element method. In *Symposium on Vibration of Civil Engineering*, 1965.
- [32] O.C. Zienkiewicz and R.E. Newton. Coupled vibrations of a structure submerged in a compressible fluid. In *Proc. Internat. Symp. on Finite Element Techniques*, pages 359–379, 1969.